

Table of Contents

- **Preface**
- **Chapter 1: The Basics**
- **Chapter 2: Elements of BASIC Language**
- **Chapter 3: Operators**
- **Chapter 4: Control Structures**
- **Chapter 5: Built-in and Library Routines**
- **Chapter 6: Examples with PIC Integrated Peripherals**
- **Chapter 7: Examples with Displaying Data**
- **Chapter 8: Examples with Memory and Storage Media**
- **Chapter 9: Communications Examples (under construction)**
- **Appendix A: mikroBasic IDE**

Preface

In order to simplify things and crash some prejudices, I will allow myself to give you some advice before reading this book. You should start reading it from the chapter that interests you the most, in any order you find suitable. As the time goes by, read the parts you may need at that exact moment. If something starts functioning without you knowing exactly how, it shouldn't bother you too much. Anyway, it is better that your program works than that it doesn't. Always stick to the practical side of life. Better to finish the application on time, make it reliable and, of course, get paid for it as well as possible.

In other words, it doesn't matter if the exact manner in which the electrons move

within the PN junctions escapes your knowledge. You are not supposed to know the whole history of electronics in order to assure the income for you or your family. Do not expect that you will find everything you need in a single book, though. The information are dispersed literally everywhere around us, so it is necessary to collect them diligently and sort them out carefully. If you do so, the success is inevitable.

With all my hopes of having done something worthy investing your time in.

Yours,
Nebojsa Matic

mikroElektronika Recommends



EasyPIC 2

Development system for PIC MCUs

USB programmer on board! System supports 18, 28 and 40-pin microcontrollers (it is delivered with PIC16F877). With the system also comes the programmer. You can test many different industrial applications on the system: temperature controllers, counters, timers... [\[more\]](#)



mikroBasic

Advanced BASIC compiler for PIC

A beginner? Worry not. Easy-to-learn BASIC syntax, advanced compiler features, built-in routines, source-level debugger, and many practical examples we have provided allow quick start in programming PIC. Highly intuitive, user-friendly IDE and comprehensive help guarantee success! [\[more\]](#)



USB PIC Flash

Programmer for PIC18 family

PIC Flash is the USB In-System programmer for Flash family of Microchip's MCUs. Beside standard FLASH MCUs, it can also program the latest models of PIC18 family. [\[more\]](#)



"PIC Microcontrollers"

On-line book, 3rd edition

The purpose of the book is not to make a microcontroller expert out of you, but to make you equal to those who had somebody to ask. Many practical examples allow quick start in programming PIC.

[\[more\]](#)

To Reader's Knowledge

The contents published in the book "Programming PIC microcontrollers in BASIC" is subject to copyright and it must not be reproduced in any form without an explicit written permission released from the editorial of mikroElektronika. The contact address for the authorization regarding contents of this book:

office@mikroelektronika.co.yu.

The book was prepared with due care and attention, however the publisher does not accept any responsibility neither for the exactness of the information published therein, nor for any consequences of its application.

PIC, PIC, PICmicro, and MPLAB are registered and protected trademarks of the Microchip Technology Inc. USA. Microchip logo and name are the registered tokens of the Microchip Technology. mikroBasic is a registered trade mark of mikroElektronika. All other tokens mentioned in the book are the property of the companies to which they belong.

mikroElektronika © 1998 - 2004. All rights reserved. If you have any questions, please contact our **office**.

Chapter 1: The Basics

- Introduction
- 1.1 Why BASIC?
- 1.2 Choosing the right PIC for the task
- 1.3 A word about code writing
- 1.4 Writing and compiling your program
- 1.5 Loading program to microcontroller
- 1.6 Running the program
- 1.7 Troubleshooting

Introduction

Simplicity and ease which higher programming languages bring in, as well as broad application of microcontrollers today, were reasons to incite some companies to adjust and upgrade BASIC programming language to better suit needs of microcontroller programming. What did we thereby get? First of all, developing applications is faster and easier with all the predefined routines which BASIC brings in, whose programming in assembly would take the largest amount of time. This allows programmer to concentrate on solving the important tasks without wasting his time on, say, code for printing on LCD display.

To avoid any confusion in the further text, we need to clarify several terms we will be using frequently throughout the book:

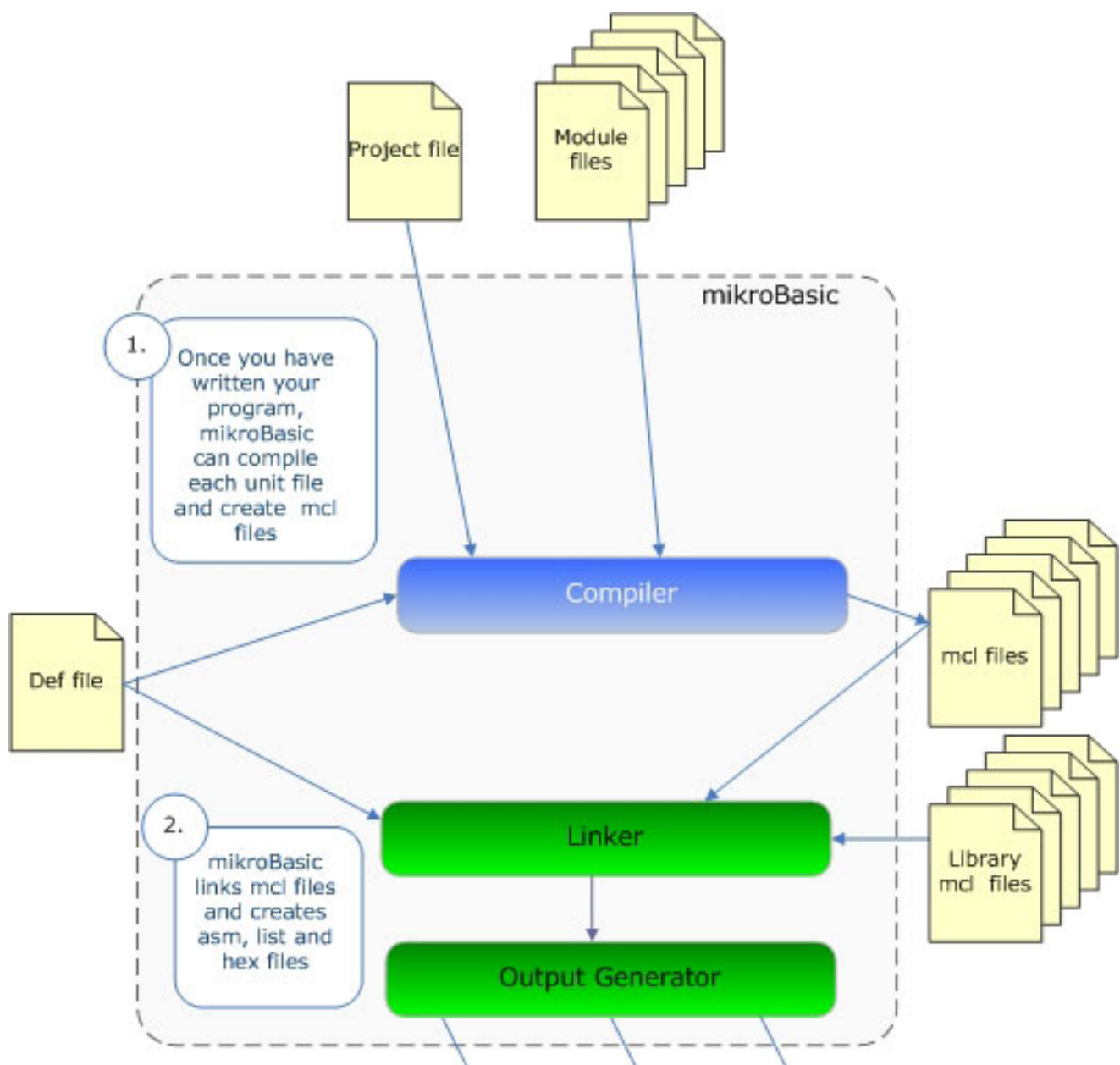
Programming language is a set of commands and rules according to which we write the program. There are various programming languages such as BASIC, C, Pascal, etc. There is plenty of resources on BASIC programming language out there, so we will focus our

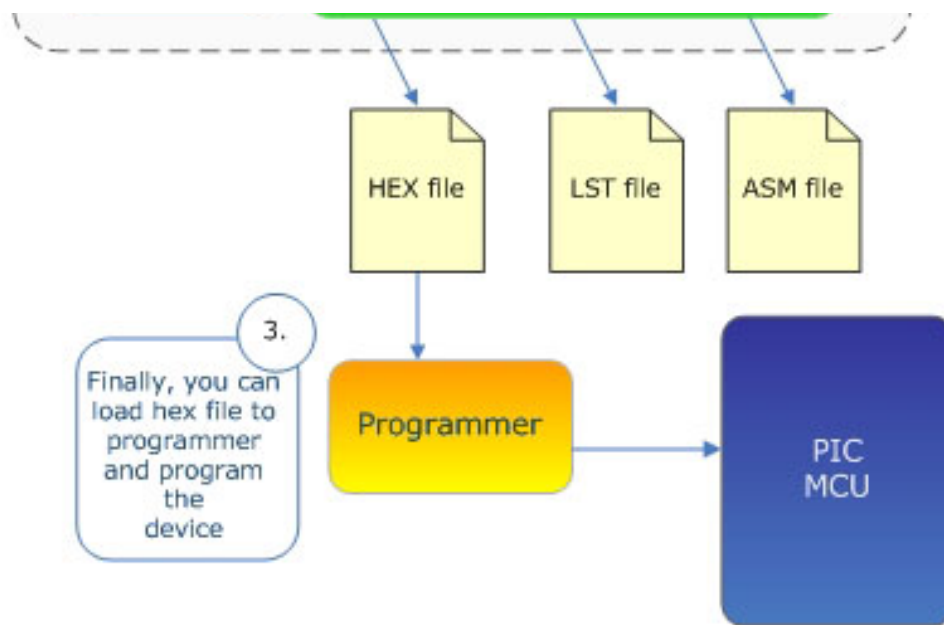
attention particularly to programming of microcontrollers.

Program consists of a sequence of commands written in programming language that microcontroller executes one after another. Chapter 2 deals with the structure of BASIC program in details.

Compiler is a program run on computer and its task is to translate the original BASIC code into language of zeros and ones that can be fed to microcontroller. The process of translation of BASIC program into executive HEX code is shown in the figure below. The program written in BASIC and saved as file `program.pbas` is converted by compiler into assembly code (`program.asm`). The generated assembly code is further translated into executive HEX code which can be written to microcontroller memory.

Programmer is a device which we use to transfer our HEX files from computer to microcontroller memory.





1.1 Why BASIC?

Originally devised as an easy-to-use tool, BASIC became widespread on home microcomputers in the 1980s, and remains popular to this day in a handful of heavily evolved dialects. BASIC's name, coined in classic, computer science tradition to produce a nice acronym, stands for *Beginner's All-purpose Symbolic Instruction Code*.

BASIC is still considered by many PC users to be the easiest programming language to use. Nowadays, this reputation is being shifted to the world of microcontrollers. BASIC allows faster and much easier development of applications for PIC compared to the Microchip's assembly language MPASM. When writing the code for MCUs, programmers frequently deal with the same issues, such as serial communication, printing on LCD display, generating PWM signals, etc. For the purpose of facilitating programming, BASIC provides a number of built-in and library routines intended for solving these problems.

As far as the execution and program size are in question, MPASM has a small advantage in respect with BASIC. This is why there is an option of combining BASIC and assembly code — assembly is commonly used for parts of program in which execution time is critical or same commands are executed great number of times. Modern microcontrollers, such as PIC, execute instructions in a single cycle. If microcontroller clock is 4MHz, then one assembly instruction requires $250\text{ns} \times 4 = 1\mu\text{s}$. As each BASIC command is technically a sequence of assembly instructions, the exact time necessary for its execution can be calculated by simply summing up the execution times of constituent assembly instructions.

1.2 Choosing the right PIC for the task

Currently, the best choice for application development using BASIC are: the famous PIC16F84, PIC16F87x, PIC16F62x, PIC18Fxxx. These controllers have program memory built on FLASH technology which provides fast erasing and reprogramming, thus allowing fast debugging. By a single mouse click in the programming software, microcontroller program can be instantly erased and then reloaded without removing chip from device. Also, program loaded in FLASH memory can be stored after the power is off. Beside FLASH memory, microcontrollers of PIC16F87x and PIC16F84 series also contain 64-256 bytes of internal EEPROM memory, which can be used for storing program data and other parameters when power is off. BASIC features built-in `EEPROM_Read` and `EEPROM_Write` instructions that can be used for loading and saving data to EEPROM.

Older PIC microcontroller families (12C67x, 14C000, 16C55x, 16C6xx, 16C7xx, and 16C92x) have program memory built on EPROM/ROM technology, so they can either be programmed only once (OTP version with ROM memory) or have a glass window (JW version with EPROM memory) which allows erasing by few minutes exposure to UV light. OTP versions are usually cheaper and are a natural choice for manufacturing large series of products.

In order to have complete information about specific microcontroller in the application, you should get the appropriate Data Sheet or Microchip CD-ROM.



The program examples worked out throughout the book are mostly to be run on the microcontrollers PIC16F84 or PIC16F877, but with minor adjustments, can be run on any other PIC microcontroller.

1.3 A word about code writing

Technically, any text editor that can save program file as pure ASCII text (without special symbols for formatting) can be used for writing your BASIC code. Still, there is no need to do it “by hand” — there are specialized environments that take care of the code syntax, free the memory and provide all the necessary tools for writing a program.

mikroBasic IDE includes highly adaptable Code Editor, fashioned to satisfy needs of both novice users and experienced programmers. Syntax Highlighting, Code Templates, Code & Parameter Assistant, Auto Correct for common typos, and other features provide comfortable environment for writing a program.

If you had no previous experience with advanced IDEs, you may wonder what Code and Parameter Assistants do. These are utilities which facilitate the code writing. For example, if you type first few letter of a word in Code Editor and then press CTRL+SPACE, all valid identifiers matching the letters you typed will be prompted to you in a floating panel. Now you can keep typing to narrow the choice, or you can select one from the list using keyboard arrows and Enter.

In combination with comprehensive help, integrated tools, extensive libraries, and Code Explorer which allows you to easily monitor program items, all the necessary tools are at hand.

1.4 Writing and compiling your program

The first step is to write our code. Every source file is saved in a single text file with extension `.pbas`. Here is an example of one simple BASIC program, `blink.pbas`.

```

program LED_Blink

main:

    TRISB = 0                ' Configure pins of PORTB as
output
    eloop:
        PORTB = $FF          ' Turn on diodes on PORTB
        Delay_ms(1000)        ' Wait 1 second
        PORTB = 0            ' Turn off diodes on PORTB
        Delay_ms(1000)        ' Wait 1 second
    goto eloop                ' Stay in loop

```

```
end.
```

When the program is completed and saved as `.pbas` file, it can be compiled by clicking on Compile Icon (or just hit CTRL+F9) in mikroBasic IDE. The compiling procedure takes place in two consecutive steps:

1. Compiler will convert `.pbas` file to assembly code and save it as `blink.asm` file.
2. Then, compiler automatically calls assembly, which converts `.asm` file into executable HEX code ready for feeding to microcontroller.

You cannot actually make the difference between the two steps, as the process is completely automated and indivisible. In case of syntax error in program code, program will not be compiled and HEX file will not be generated. Errors need to be corrected in the original `.pbas` file and then the source file may be compiled again. The best approach is to write and test small, logical parts of the program to make debugging easier.

1.5 Loading program to microcontroller

As a result of successful compiling of our previous code, mikroBasic will generate following files:

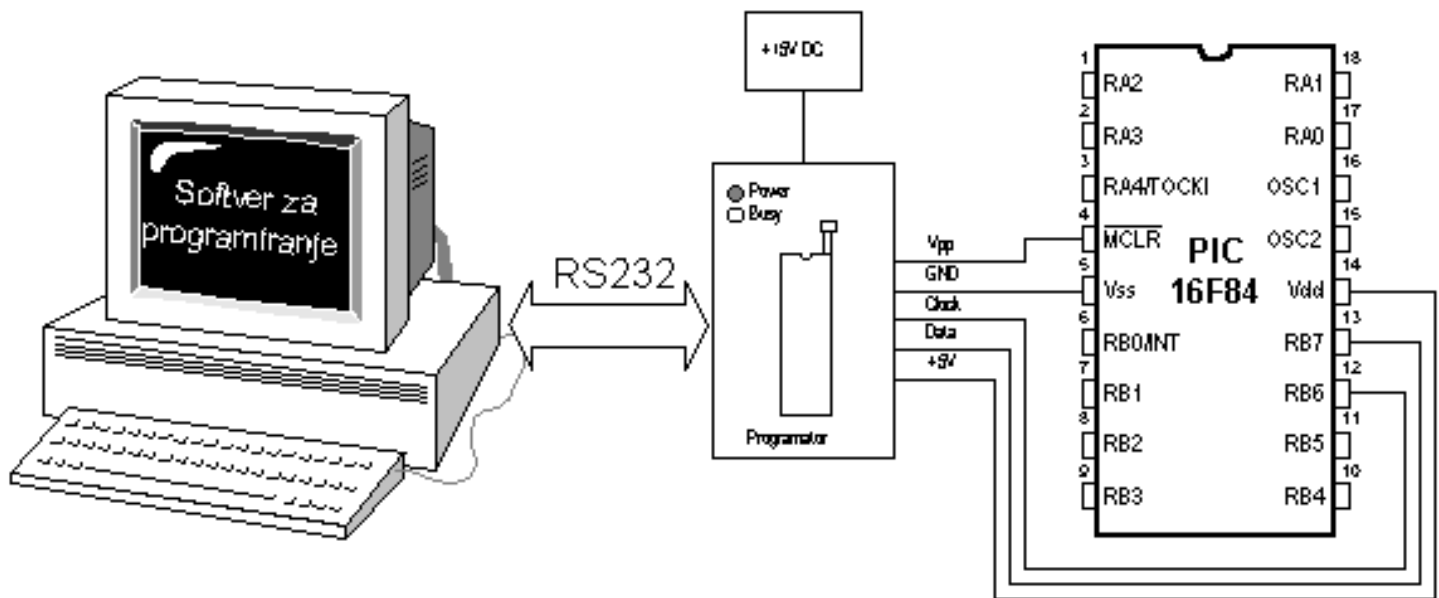
- `blink.asm` - assembly file
- `blink.lst` - program listing
- `blink.mcl` - mikro compile library
- `blink.hex` - executable file which is written into the programming memory

MCL file (mikro compile library) is created for each module you have included in the project. In the process of compiling, `.mcl` files will be linked together to output `asm`, `lst` and `hex` files. If you want to distribute your module without disclosing the source code, you can send your compiled library (file extension `.mcl`). User will be able to use your library as if he had the source code. Although the compiler is able to determine which routines are implemented in the library, it is a common practice to provide routine prototypes in a separate text file.

HEX file is the one you need to program the microcontroller. Commonly, generated HEX will be standard 8-bit Merged Intel HEX format, accepted by the vast majority of the programming software. The programming device (programmer) with accessory software installed on PC is in charge of writing the physical contents of HEX file into the internal memory of a microcontroller. The contents of a file `blink.hex` is given below:

```
:100000000428FF3FFF3FFF3F031383168601FF30A5
:10001000831286000630F000FF30F100FF30F2005E
:10002000F00B13281A28F10B16281928F20B1628A2
:10003000132810281A30F000FF30F100F00B2128AF
:100040002428F10B21281E284230F000F00B26282E
:1000500086010630F000FF30F100FF30F200F00BB7
:1000600032283928F10B35283828F20B3528322868
:100070002F281A30F000FF30F100F00B4028432801
:10008000F10B40283D284230F000F00B45280428B1
:100090004828FF3FFF3FFF3FFF3FFF3FFF3FFF3F3E
:02400E007A3FF7
:00000001FF
```

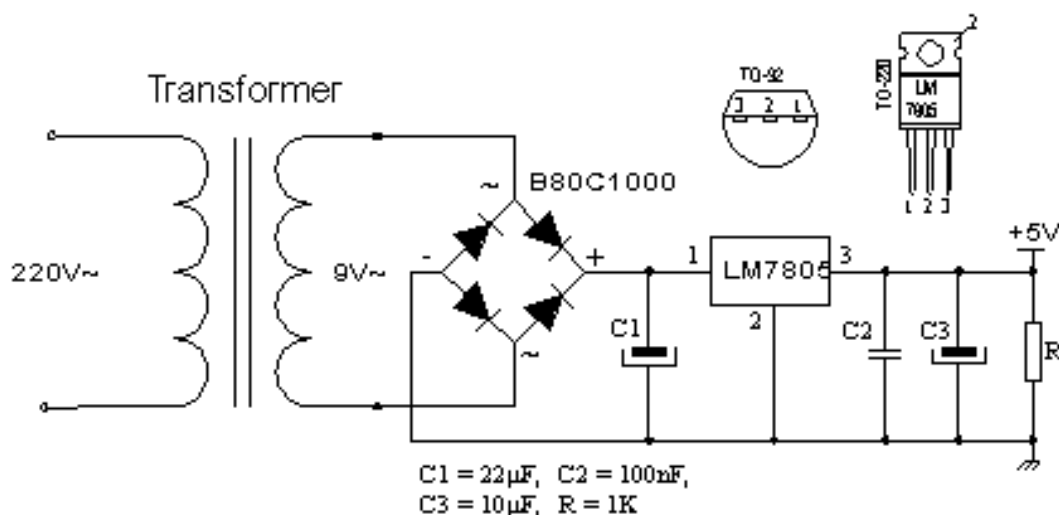
Beside loading a program code into programming memory, programmer also configures the target microcontroller, including the type of oscillator, protection of memory against reading, watchdog timer, etc. The following figure shows the connection between PC, programming device and the MCU.



Note that the programming software should be used *only* for the communication with the programming device — it is not suitable for code writing.

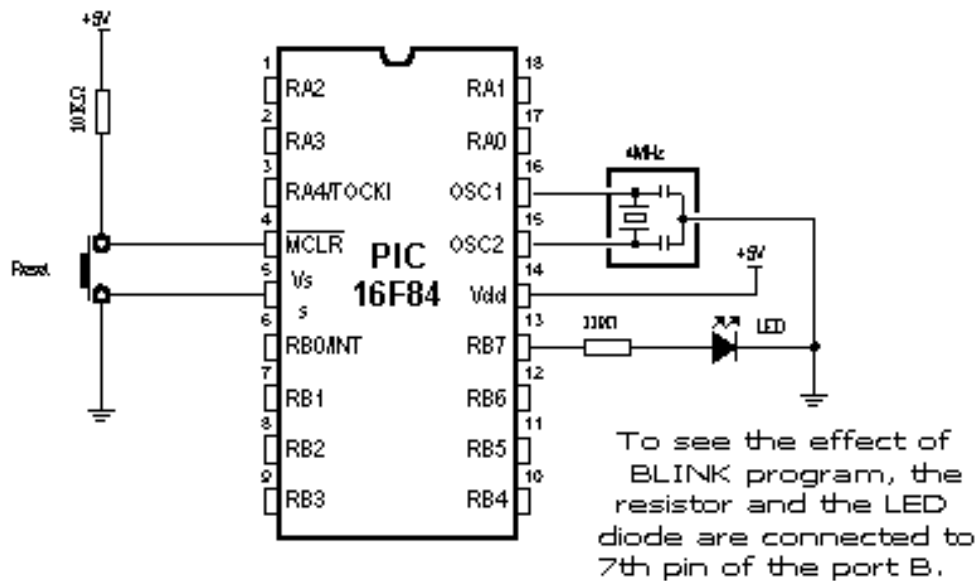
1.6 Running the program

For proper functioning of microcontroller, it is necessary to provide **power supply**, **oscillator**, and a **reset circuit**. The supply can be set with the simple rectifier with Gretz junction and LM7805 circuit as shown in the figure below.



Oscillator can be 4MHz crystal and either two 22pF capacitors or the ceramic resonator of the same frequency (ceramic resonator already contains the mentioned capacitors, but unlike oscillator has three termination instead of only two). The rate at which the microcontroller operates, i.e. the speed at which the program runs, depends heavily on the oscillator frequency. During the application development, the easiest thing to do is to use the internal

reset circuit — MCLR pin is connected to +5V through a 10K resistor. Below is the scheme of a rectifier with LM7805 circuit which gives the output of stable +5V, and the minimal configuration relevant for the operation of a PIC microcontroller.



After the supply is brought to the circuit previously shown, PIC microcontroller should look animated, and the LED diode should blink once every second. If the signal is completely missing (LED diode does not blink), then check if +5V is present at all the relevant pins of PIC.

1.7 Troubleshooting

There are several commonly encountered problems of bringing PIC microcontroller to working conditions. You need to check a few external components and test whether their values correspond to the desired ones, and finally to see whether all the connections are done right. We will present a few notes you may find useful.

- Check whether the MCLR pin is connected to +5V, over reset circuit, or simply with 10K resistor. If the pin remains disconnected, its level will be “floating” and it may work sometimes, but it usually won’t. Chip has power-on-reset circuit, so the appropriate external pull-up resistor on MCLR pin should be sufficient.
- Check whether the connection with the resonator is stable. For most PIC microcontrollers to begin with 4MHz resonator is well enough.
- Check the supply. PIC microcontroller consumes very little energy but the supply needs to be well filtrated. At the rectifier output, the current is direct but pulsating,

and as such is not suitable for the supply of microcontroller. To avoid the pulsating, the electrolytic capacitor of high capacitance (e.g. 470 mF) is placed at the rectifier output.

- If PIC microcontroller supervises devices that pull a lot of energy, they may provoke enough malfunctioning on the supply lines to cause the microcontroller start behaving somewhat strangely. Even seven-segmented LED display may well induce tension drops (the worst scenario is when all the digits are 8, when LED display needs the most power), if the source itself is not capable to procure enough current (e.g. 9V battery).
- Some PIC microcontrollers feature multi-functional I/O pins, for example PIC16C62x family (PIC16C620, 621 and 622). Controllers of this family are provided with analog comparators on port A. After putting those chips to work, port A is set to analog mode, which brings about the unexpected behavior of the pin functions on the port. Upon reset, any PIC with analog inputs will show itself in analog mode (if the same pins are used as digital lines they need to be set to digital mode). One possible source of troubles is that the fourth pin of port A exhibits singular behavior when it is used as output, because the pin has open collectors output instead of usual bipolar state. This implies that clearing this pin will nevertheless set it to low level, while setting the pin will let it float somewhere in between, instead of setting it to high level. To make the pin behave as expected, the pull-up resistor was placed between RA4 and 5V. Its magnitude is between 4.7K and 10K, depending on the current necessary for the input. In this way, the pin functions as any other input pin (all pins are output after reset).

More problems are to be expected if you plan to be seriously working with PIC. Sometimes the thing seems like it is going to work, but it just won't, regardless of the effort. Just remember that there is always more than one way to solve the problem, and that a different approach may bring solution.

PIC, PIC, PICmicro, and MPLAB are registered and protected trademarks of the Microchip Technology Inc. USA. Microchip logo and name are the registered tokens of the Microchip Technology. mikroBasic is a registered trade mark of mikroElektronika. All other tokens mentioned in the book are the property of the companies to which they belong.

mikroElektronika © 1998 - 2004. All rights reserved. If you have any questions, please contact our
office.

Chapter 2: Elements of BASIC Language

- Introduction
- 2.1 Identifiers
- 2.2 Operators
- 2.3 Expressions
- 2.4 Instructions
- 2.5 Data Types
- 2.6 Constants
- 2.7 Variables
- 2.8 Symbols
- 2.9 Directives
- 2.10 Comments
- 2.11 Labels
- 2.12 Procedures and Functions
- 2.13 Modules

Introduction

This chapter deals with the elements of BASIC language and the ways to use them efficiently. Learning how to program is not complicated, but it requires skill and experience to write code that is efficient, legible, and easy to handle. First of all, program is supposed to be comprehensible, so that the programmer himself, or somebody else working on the application, could make necessary corrections and improvements. We have provided a code sample written in a clear and manifest way to give you an idea how programs could be written:

```
' *****
' microcontroller : P16F877A
'
' Project: Led_blinking
' This project is designed to work with PIC 16F877A;
' with minor adjustments, it should work with any other PIC MCU.
'
' The code demonstrates blinking of diodes connected to PORTB.
' Diodes go on and off each second.
' *****
```

```

program LED_Blinking

main:                ' Beginning of program
    TRISB = 0        ' Configure pins of PORTB as output
    PORTB = %11111111 ' Turn ON diodes on PORTB
    Delay_ms(1000)    ' Wait for 1 second
    PORTB = %00000000 ' Turn OFF diodes on PORTB
    Delay_ms(1000)    ' Wait for 1 second
    goto main         ' Endless loop
end.                ' End of program

```

Through clever use of comments, symbols, labels and other elements supported by BASIC, program can be rendered considerably clearer and more understandable, offering programmer a great deal of help.

Also, it is advisable to divide larger programs into separate logical entities (such as routines and modules, see below) which can be addressed when needed. This also increases reusability of code.

Names of routines and labels indicating a program segment should make some obvious sense. For example, program segment that swaps values of 2 variables, could be named "Swap", etc.

2.1 Identifiers

Identifiers are names used for referencing the stored values, such as variables and constants. Every program, module, procedure, and function must be identified (hence the term) by an identifier.

Valid identifier:

1. Must begin with a letter of English alphabet or possibly the underscore (_)
2. Consists of alphanumeric characters and the underscore (_)
3. May not contain special characters:
~ ! @ # \$ % ^ & * () + ` - = { } [] : " ; ' < > ? , . / | \
4. Can be written in mixed case as BASIC is case insensitive; e.g. `First`, `FIRST`, and `fIrST` are an equivalent identifier.

Elements ignored by the compiler include spaces, new lines, and tabs. All these elements are collectively known as the “white space”. White space serves only to make the code more legible – it does not affect the actual compiling.

Several identifiers are reserved in BASIC, meaning that you cannot use them as your own identifiers (e.g. words `function`, `byte`, `if`, etc). For more information, please refer to the list of reserved words. Also, BASIC has a number of predefined identifiers which are listed in Chapter 4: Instructions.

2.2 Operators

BASIC language possesses set of operators which is used to assign values, compare values, and perform other operations. The objects manipulated for that purpose are called operands (which themselves can be variables, constants, or other elements).

Operators in BASIC must have at least two operands, with an exception of two unary operators. They serve to create expressions and instructions that in effect compose the program.

There are four types of operators in BASIC:

1. Arithmetic Operators
2. Boolean Operators
3. Logical (Bitwise) Operators
4. Relation Operators (Comparison Operators)

Operators are covered in detail in chapter 3.

2.3 Expressions

Expression is a construction that returns a value. BASIC syntax restricts you to single line expressions, where *carriage return* character marks the end of the expression. The simplest expressions are variables and constants, while more complex can be constructed from simpler ones using operators, function calls, indexes, and typecasts. Here is one simple expression:

```
A = B + C      ' This expression sums up the values of variables B and C
               ' and stores the result into variable A.
```

You need to pay attention that the sum must be within the range of variable A in order to avoid the overflow and therefore the evident computational error. If the result of the expression amounts to 428, and the variable A is of byte type (having range between 0 and 255), the result accordingly obtained will be 172, which is obviously wrong.

2.4 Instructions

Each instruction determines an action to be performed. As a rule, instructions are being executed in an exact order in which they are written in the program. However, the order of their execution can be changed by means of jump, routine call, or an interrupt.

```
if Time = 60 then
```

```

goto Minute      ' If variable Time equals 60 jump to label Minute
end if

```

Instruction `if...then` contains the conducting expression `Time = 60` composed of two operands, variable `Time`, constant `60` and the comparison operator (`=`). Generally, instructions may be divided into **conditional instructions** (decision making), **loops** (repeating blocks), **jumps**, and specific **built-in instructions** (e.g. for accessing the peripherals of microcontroller). Instruction set is explained in detail in Chapter 4: Instructions.

2.5 Data Types

Type determines the allowed range of values for variable, and which operations may be performed on it. It also determines the amount of memory used for one instance of that variable.

Simple data types include:

Type	Size	Range of values
byte	8-bit	0 .. 255
char*	8-bit	0 .. 255
word	16-bit	0 .. 65535
short	16-bit	-128 .. 127
integer	16-bit	-32768 .. 32767
longint	32-bit	-2147483648 .. 2147483647

* char type can be treated as byte type in every aspect

Structured types include:

Array, which represent an indexed collection of elements of the same type, often called the base type. Base type can be any simple type.

String represents a sequence of characters. It is an array that holds characters and the first element of string holds the number of characters (max number is 255).

Sign is important attribute of data types, and affects the way variable is treated by the compiler.

Unsigned can hold only positive numbers:

```

byte      0 .. 255
word     0 .. 65535

```

Signed can hold both positive and negative numbers:

```
short      -128 .. 127
integer    -32768 .. 32767
longint    -2147483648 .. 214748364
```

2.6 Constants

Constant is data whose value cannot be changed during the runtime. Every constant is declared under unique name which must be a valid identifier. It is a good practice to write constant names in uppercase.

If you frequently use the same fixed value throughout the program, you should declare it a constant (for example, maximum number allowed is 1000). This is a good practice since the value can be changed simply by modifying the declaration, instead of going through the entire program and adjusting each instance manually. As simple as this:

```
const MAX = 1000
```

Constants can be declared in decimal, hex, or binary form. Decimal constants are written without any prefix. Hexadecimal constants begin with a sign \$, while binary begin with %.

```
const A = 56           ' 56 decimal
const B = $0F          ' 15 hexadecimal
const C = %10001100    ' 140 binary
```

It is important to understand why constants should be used and how this affects the MCU. Using a constant in a program consumes no RAM memory. This is very important due to the limited RAM space (PIC16F877 has 368 locations/bytes).

2.7 Variables

Variable is data whose value can be changed during the runtime. Each variable is declared under unique name which has to be a valid identifier. This name is used for accessing the memory location occupied by the variable. Variable can be seen as a container for data and because it is typed, it instructs the compiler how to interpret the data it holds.

In BASIC, variable needs to be declared before it can be used. Specifying a data type for each variable is mandatory. Variable is declared like this:

```
dim identifier as type
```

where *identifier* is any valid identifier and *type* can be any given data type.

For example:

```
dim temperature as byte      ' Declare variable temperature of byte type
dim voltage as word          ' Declare variable voltage of word type
```

Individual bits of byte variables (including SFR registers such as PORTA, etc) can be accessed by means of dot, both on left and right side of the expression. For example:

```
Data_Port.3 = 1                ' Set third bit of byte variable Data_Port
```

2.8 Symbols

Symbol makes possible to replace a certain expression with a single identifier alias. Use of symbols can increase readability of code.

BASIC syntax restricts you to single line expressions, allowing shortcuts for constants, simple statements, function calls, etc. Scope of symbol identifier is a whole source file in which it is declared.

For example:

```
symbol MaxAllowed = 234          ' Symbol as alias for numeric value
symbol PORT = PORTC              ' Symbol as alias for Special Function
Register
symbol DELAY1S = Delay_ms(1000)  ' Symbol as alias for procedure call
...
if teA > MaxAllowed then
    teA = teA - 100
```

```

end if
PORT.1  = 0
DELAY1S
...

```

Note that using a symbol in a program technically consumes no RAM memory – compiler simply replaces each instance of a symbol with the appropriate code from the declaration.

2.9 Directives

Directives are words of special significance for BASIC, but unlike other reserved words, appear only in contexts where user-defined identifiers cannot occur. You cannot define an identifier that looks exactly like a directive.

Directive	Meaning
Absolute	specify exact location of variable in RAM
Org	specify exact location of routine in ROM

Absolute specifies the starting address in RAM for variable (if variable is multi-byte, higher bytes are stored at consecutive locations).

Directive `absolute` is appended to the declaration of variable:

```

dim rem as byte absolute $22
' Variable will occupy 1 byte at address $22

dim dot as word absolute $23
' Variable will occupy 2 bytes at addresses $23 and $24

```

Org specifies the starting address of routine in ROM. For PIC16 family, routine must fit in one page – otherwise, compiler will report an error. Directive `org` is appended to the declaration of routine:

```

sub procedure test org $200
' Procedure will start at address $200
...
end sub

```


2.10 Comments

Comments are text that is added to the code for purpose of description or clarification, and are completely ignored by the compiler.

```
' Any text between an apostrophe and the end of the  
' line constitutes a comment. May span one line only.
```

It is a good practice to comment your code, so that you or anybody else can later reuse it. On the other hand, it is often useful to comment out a troublesome part of the code, so it could be repaired or modified later. Comments should give purposeful information on what the program is doing. Comment such as *Set Pin0* simply explains the syntax but fails to state the purpose of instruction. Something like *Turn Relay on* might prove to be much more useful.

Specialized editors feature syntax highlighting – it is easy to distinguish comments from code due to different color, and comments are usually italicized.

2.11 Labels

Labels represent the most direct way of controlling the program flow. When you mark a certain program line with label, you can jump to that line by means of instructions `goto` and `gosub`. It is convenient to think of labels as bookmarks of sort. Note that the label `main` must be declared in every BASIC program because it marks the beginning of the main module.

Label name needs to be a valid identifier. You cannot declare two labels with same name within the same routine. The scope of label (label visibility) is tied to the routine where it is declared. This ensures that `goto` cannot be used for jumping between routines.

`Goto` is an unconditional jump statement. It jumps to the specified label and the program execution continues normally from that point on.

`Gosub` is a jump statement similar to `goto`, except it is tied to a matching word `return`. Upon jumping to a specified label, previous address is saved on the stack. Program will continue executing normally from the label, until it reaches `return` statement – this will exit the subroutine and return to the first program line following the caller `gosub` instruction.

Here is a simple example:

```

program test

main:

' some instructions...

' simple endless loop using a label

my_loop:

' some instructions...

' now jump back to label _loop

goto my_loop

end.

```

Note: Although it might seem like a good idea to beginners to program by means of jumps and labels, you should try not to depend on it. This way of thinking strays from the procedural programming and can teach you bad programming habits. It is far better to use procedures and functions where applicable, making the code structure more legible and easier to maintain.

2.12 Procedures and Functions

Procedures and functions, referred to as routines, are self-contained statement blocks that can be called from different locations in a program. Function is a routine that returns a value upon execution. Procedure is a routine that does not return a value.

Once routines have been defined, you can call them any number of times. Procedure is called upon to perform a certain task, while function is called to compute a certain value.

Procedure declaration has the form:

```

sub procedure procedureName(parameterList)
    localDeclarations
    statements
end sub

```

where *procedureName* is any valid identifier, *statements* is a sequence of statements that are executed upon the

calling the procedure, and (*parameterList*), and *localDeclarations* are optional declaration of variables and/or constants.

```
sub procedure pr1_procedure(dim par1 as byte, dim par2 as byte,
                           dim byref vp1 as byte, dim byref vp2 as byte)
dim locS as byte
  par1 = locS + par1 + par2
  vp1  = par1 or par2
  vp2  = locS xor par1
end sub
```

par1 and *par2* are passed to the procedure by the value, but variables marked by keyword **byref** are passed by the address.

This means that the procedure call

```
pr1_procedure(tA, tB, tC, tD)
```

passes tA and tB by the value: creates *par1* = tA; and *par2* = tB; then manipulates *par1* and *par2* so that tA and tB remain unchanged;

passes tC and tD by the address: whatever changes are made upon *vp1* and *vp2* are also made upon tC and tD.

Function declaration is similar to procedure declaration, except it has a specified return type and a return value. Function declaration has the form:

```
sub function functionName(parameterList) as returnType
  localDeclarations
  statements
end sub
```

where *functionName* is any valid identifier, *returnType* is any simple type, *statements* is a sequence of statements to be executed upon calling the function, and (*parameterList*), and *localDeclarations* are optional declaration of variables and/or constants.

In BASIC, we use the keyword **Result** to assign return value of a function. For example:

```
sub function Calc(dim par1 as byte, dim par2 as word) as word
dim locS as word
```

```

locS = par1 * (par2 + 1)
Result = locS
end sub

```

As functions return values, function calls are technically expressions. For example, if you have defined a function called `Calc`, which collects two integer arguments and returns an integer, then the function call `Calc(24, 47)` is an integer expression. If `I` and `J` are integer variables, then `I + Calc(J, 8)` is also an integer expression.

2.13 Modules

Large programs can be divided into *modules* which allow easier maintenance of code. Each module is an actual file, which can be compiled separately; compiled modules are linked to create an application. Note that each source file must end with keyword `end` followed by a dot.

Modules allow you to:

1. Break large code into segments that can be edited separately,
2. Create libraries that can be used in different programs,
3. Distribute libraries to other developers without disclosing the source code.

In mikroBasic IDE, all source code including the main program is stored in `.pbas` files. Each project consists of a single project file, and one or more module files. To build a project, compiler needs either a source file or a compiled file for each module.

Every BASIC application has one main module file and any number of additional module files. All source files have same extension (`pbas`). Main file is identified by keyword `program` at the beginning, while other files have keyword `module` instead. If you want to include a module, add the keyword `include` followed by a quoted name of the file.

For example:

```

program test_project
include "math2.pbas"
dim tA as word
dim tB as word

main:
    tA = sqrt(tb)
end.

```

Keyword `include` instructs the compiler which file to compile. The example above includes module `math2.pbas` in the program file. Obviously, routine `sqrt` used in the example is declared in module `math2.pbas`.

If you want to distribute your module without disclosing the source code, you can send your compiled library (file extension `.mcl`). User will be able to use your library as if he had the source code. Although the compiler is able to determine which routines are implemented in the library, it is a common practice to provide routine prototypes in a separate text file.

Module files should be organized in the following manner:

```

module unit_name           ' Module name
include ...                ' Include other modules if necessary

symbol ...                ' Symbols declaration
const ...                 ' Constants declaration
dim ...                   ' Variables declaration

sub procedure procedure_name ' Procedures declaration
    ...
end sub

sub function function_name  ' Functions declaration
    ...
end sub

end.                      ' End of module

```

Note that there is no “body” section in the module – module files serve to declare functions, procedures, constants and global variables.

PIC, PIC, PICmicro, and MPLAB are registered and protected trademarks of the Microchip Technology Inc. USA. Microchip logo and name are the registered tokens of the Microchip Technology. mikroBasic is a registered trade mark of mikroElektronika. All other tokens mentioned in the book are the property of the companies to which they belong.

mikroElektronika © 1998 - 2004. All rights reserved. If you have any questions, please contact our **office**.

Chapter 3: Operators

- Introduction
- 3.1 Arithmetic Operators
- 3.2 Boolean Operators
- 3.3 Logical (Bitwise) Operators
- 3.4 Relation Operators (Comparison Operators)

Introduction

In complex expressions, operators with higher precedence are evaluated before the operators with lower precedence; operators of equal precedence are evaluated according to their position in the expression starting from the left.

Operator	Priority
not	first (highest)
*, div, mod, and, <<, >>	second
+, -, or, xor	third
=, <>, <, >, <=, >=	fourth (lowest)

3.1 Arithmetic Operators

Overview of arithmetic operators in BASIC:

Operator	Operation	Operand types	Result type
+	addition	byte, short, integer, words, longint	byte, short, integer, words, longint
-	subtraction	byte, short, integer, words, longint	byte, short, integer, words, longint
*	multiplication	byte, short, integer, words	integer, words, long
div	division	byte, short, integer, words	byte, short, integer, words
mod	remainder	byte, short, integer, words	byte, short, integer, words

A **div** B is the value of A divided by B rounded down to the nearest integer. The **mod** operator returns the remainder obtained by dividing its operands. In other words,

$$X \text{ mod } Y = X - (X \text{ div } Y) * Y.$$

If 0 (zero) is used explicitly as the second operand (i.e. X div 0), compiler will report an error and will not generate code. But in case of implicit division by zero : X div Y , where Y is 0 (zero), result will be the maximum value for the appropriate type (for example, if X and Y are words, the result will be \$FFFF).

If number is converted from less complex to more complex data type, upper bytes are filled with zeros. If number is converted from more complex to less complex data type, data is simply truncated (upper bytes are lost).

If number is converted from less complex to more complex data type, upper bytes are filled with ones if sign bit equals 1 (number is negative). Upper bytes are filled with zeros if sign bit equals 0 (number is positive). If number is converted from more complex to less complex data type, data is simply truncated (upper bytes are lost).

BASIC also has two unary arithmetic operators:

Operator	Operation	Operand types	Result type
+ (unary)	sign identity	short, integer, longint	short, integer, longint
- (unary)	sign negation	short, integer, longint	short, integer, longint

Unary arithmetic operators can be used to change sign of variables:

```
a = 3
b = -a
' assign value -3 to b
```

3.2 Boolean Operators

Boolean operators are not true operators, because there is no boolean data type defined in BASIC. These operators conform to standard Boolean logic. They cannot be used with any data type, but only to build complex conditional expression.

Operator	Operation
not	negation
and	conjunction
or	disjunction

For example:

```

if (astr > 10) and (astr < 20) then
PORTB = 0xFF
end if

```

3.3 Logical (Bitwise) Operators

Overview of logical operators in BASIC:

Operator	Operation	Operand types	Result type
not	bitwise negation	byte, word, short, integer, long	byte, word, short, integer, long
and	bitwise and	byte, word, short, integer, long	byte, word, short, integer, long
or	bitwise or	byte, word, short, integer, long	byte, word, short, integer, long
xor	bitwise xor	byte, word, short, integer, long	byte, word, short, integer, long
<<	bit shift left	byte, word, short, integer, long	byte, word, short, integer, long
>>	bit shift right	byte, word, short, integer, long	byte, word, short, integer, long

<< : shift left the operand for a number of bit places specified in the right operand

(must be positive and less than 255).

>> : shift right the operand for a number of bit places specified in the right operand (must be positive and less than 255).

For example, if you need to extract the higher byte, you can do it like this:

```
dim temp as word

main:
  TRISA = word(temp >> 8)
end.
```

3.4 Relation Operators (Comparison Operators)

Relation operators (Comparison operators) are commonly used in conditional and loop statements for controlling the program flow. Overview of relation operators in BASIC:

Operator	Operation	Operand types	Result type
=	equality	All simple types	True or False
<>	inequality	All simple types	True or False
<	less-than	All simple types	True or False
>	greater-than	All simple types	True or False
<=	less-than-or-equal-to	All simple types	True or False
>=	greater-than-or-equal-to	All simple types	True or False

PIC, PIC, PICmicro, and MPLAB are registered and protected trademarks of the Microchip Technology Inc. USA. Microchip logo and name are the registered tokens of the Microchip Technology. mikroBasic is a registered trade mark of mikroElektronika. All other tokens mentioned in the book are the property of the companies to which they belong.

mikroElektronika © 1998 - 2004. All rights reserved. If you have any questions, please contact our **office**.

Chapter 4: Control Structures

- Introduction
- 4.1 Conditional Statements
 - 4.1.1 IF..THEN Statement
 - 4.1.2 SELECT..CASE Statement
 - 4.1.3 GOTO Statement
- 4.2 Loops
 - 4.2.1 FOR Statement
 - 4.2.2 DO..LOOP Statement
 - 4.2.3 WHILE Statement
- 4.3 ASM Statement

Introduction

Statements define algorithmic actions within a program. Simple statements - like assignments and procedure calls - can be combined to form loops, conditional statements, and other structured statements.

Simple statement does not contain any other statements. Simple statements include assignments, and calls to procedures and functions.

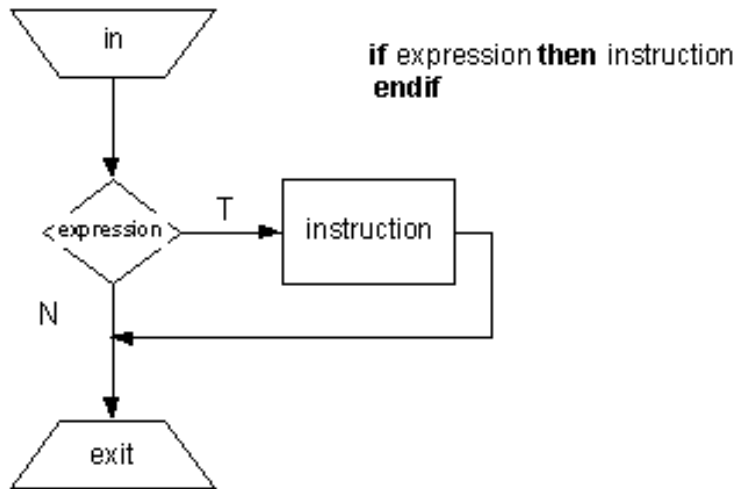
Structured statements are constructed from other statements. Use a structured statement when you want to execute other statements sequentially, conditionally, or repeatedly.

4.1 Conditional Statements

Conditional statements are used for change the flow of the program execution upon meeting a certain

condition. The BASIC instruction of branching in BASIC language is the IF instruction, with several variations that provide the necessary flexibility.

4.1.1 IF..THEN Statement – *conditional program branching*

Syntax	<pre> if expression then statements1 [else statements2] end if </pre>
Description	<p>Instruction selects one of two possible program paths. Instruction IF..THEN is the fundamental instruction of program branching in PIC BASIC and it can be used in several ways to allow flexibility necessary for realization of decision making logic.</p> <p><i>Expression</i> returns a True or False value. If <i>expression</i> is True, then <i>statements1</i> are executed; otherwise <i>statements2</i> are executed, if the <code>else</code> clause is present. <i>Statements1</i> and <i>statements2</i> can be statements of any type.</p>
Example	<p>The simplest form of the instruction is shown in the figure below. Our example tests the button connected to RB0 - when the button is pressed, program jumps to the label "Add" where value of variable "w" is increased. If the button is not pressed, program jumps back to the label "Main".</p>  <pre> dim j as byte Main: </pre>

```
if PORTB.0 = 0 then
```

```
    goto Add
```

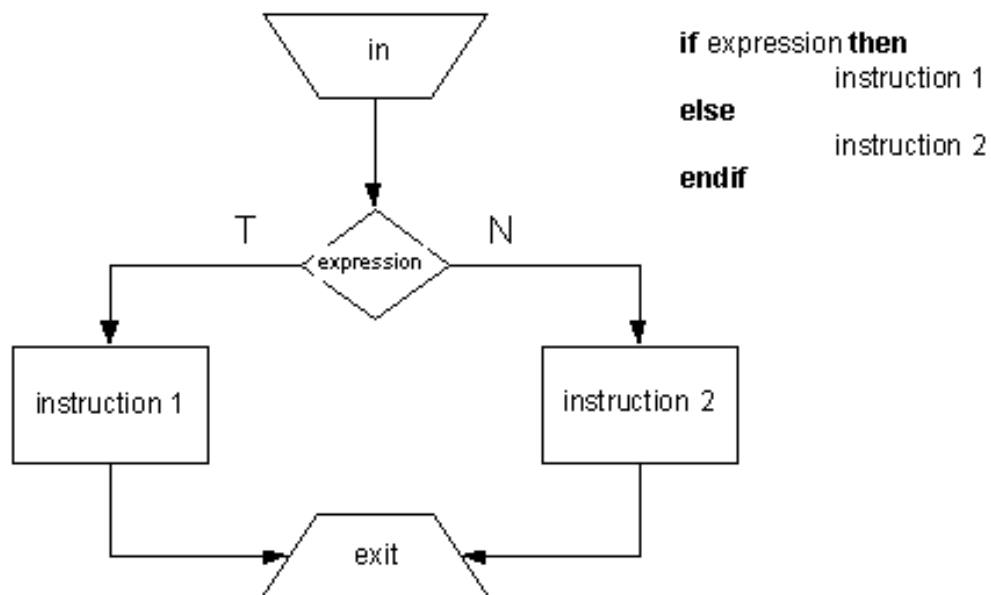
```
end if
```

```
    goto Main
```

```
Add: j = j + 1
```

```
end.
```

More complex form of instruction is program branching with the ELSE clause:



```
dim j as byte
```

```
Main:
```

```
    if PORTB.0 = 0 then
```

```
        j = j + 1
```

```
    else
```

```
        j = j - 1
```

```
    endif
```

```
    goto Main
```



```
end.
```

4.1.2 SELECT..CASE Statement – *Conditional multiple program branching*

Syntax	<pre> select case Selector case Value_1 Statements_1 case Value_2 Statements_2 ... case Value_N Statements_n [case else Statements_else] end select </pre>
Description	<p>Select Case statement is used for selecting one of several available branches in the program course. It consists of a selector variable as a switch condition, and a list of possible values. These values can be constants, numerals, or expressions.</p> <p>Eventually, there can be an else statement which is executed if none of the labels corresponds to the value of the selector.</p> <p>As soon as the Select Case statement is executed, at most one of the statements <i>statements_1 .. statements_n</i> will be executed. The <i>Value</i> which matches the <i>Selector</i> determines the statements to be executed.</p> <p>If none of the <i>Value</i> items matches the <i>Selector</i>, then the <i>statements_else</i> in the else clause (if there is one) are executed.</p>

Example	<pre> select case W case 0 B = 1 PORTB = B case 1 A = 1 PORTA = A case else PORTB = 0 end select ... select case Ident case testA PORTB = 6 Res = T mod 23 case teB + teC T = 1313 case else T = 0 end select </pre>
----------------	---

4.1.3 GOTO Statement – *Unconditional jump to the specified label*

Syntax	<code>goto Label</code>
Description	<p>Goto statement jumps to the specified label unconditionally, and the program execution continues normally from that point on.</p> <p>Avoid using GOTO too often, because over-labeled programs tend to be less intelligible.</p>

Example	<pre> program test main: ' <i>some instructions ...</i> goto myLabel ' <i>some instructions...</i> myLabel: ' <i>some instructions...</i> end. </pre>
----------------	---

4.2 Loops

Loop statements allow repeating one or more instructions for a number of times. The conducting expression determines the number of iterations loop will go through.

4.2.1 FOR Statement – *Repeating of a program segment*

Syntax	<pre> for counter = initialValue to finalValue [step step_value] statement_1 statement_2 ... statement_N next counter </pre>
---------------	--

Description	<p>For statement requires you to specify the number of iterations you want the loop to go through.</p> <p><i>Counter</i> is variable; <i>initialValue</i> and <i>finalValue</i> are expressions compatible with <i>counter</i>; <i>statement</i> is any statement that does not change the value of <i>counter</i>; <i>step_value</i> is value that is added to the <i>counter</i> in each iteration. <i>Step_value</i> is optional, and defaults to 1 if not stated otherwise. Be careful when using large values for <i>step_value</i>, as overflow may occur.</p> <p>Every statement between <code>for</code> and <code>next</code> will be executed once per iteration.</p>
Example	<p>Here is a simple example of a FOR loop used for emitting hex code on PORTB for 7-segment display with common cathode. Nine digits should be printed with one second delay.</p> <pre> for i = 1 to 9 portb = i delay_ms(1000) next i </pre>

4.2.2 DO..LOOP Statement – Loop until condition is fulfilled

Syntax	<pre> do statement_1 ... statement_N loop until expression </pre>
Description	<p><i>Expression</i> returns a True or False value. The <code>do . . loop</code> statement executes <i>statement_1</i>; ...; <i>statement_N</i> continually, checking the <i>expression</i> after each iteration. Eventually, when <i>expression</i> returns True, the <code>do . . loop</code> statement terminates.</p> <p>The sequence is executed at least once because the check takes place in the end.</p>

Example	<pre> I = 0 do I = I + 1 ' execute these 2 statements PORTB = I ' until i equals 10 (ten) loop until I = 10 </pre>
----------------	--

4.2.3 WHILE Statement – Loop while condition is fulfilled

Syntax	<pre> while expression statement_0 statement_1 ... statement_N wend </pre>
Description	<p><i>Expression</i> is tested first. If it returns True, all the following statements enclosed by while and wend will be executed (or only one statement, alternatively). It will keep on executing <i>statements</i> until the <i>expression</i> returns False.</p> <p>Eventually, as <i>expression</i> returns False, while will be terminated without executing <i>statements</i>.</p> <p>While is similar to do . . loop, except the check is performed at the beginning of the loop. If <i>expression</i> returns False upon first test, <i>statements</i> will not be executed.</p>
Example	<pre> while I < 90 I = I + 1 wend ... while I > 0 I = I div 3 PORTA = I </pre>

wend

4.3 ASM Statement – *Embeds assembly instruction block*

Syntax	<pre>asm statementList end asm</pre>
Description	<p>Sometimes it can be useful to write part of the program in assembly. ASM statement can be used to embed PIC assembly instructions into BASIC code.</p> <p>Note that you cannot use numerals as absolute addresses for SFR or GPR variables in assembly instructions. You may use symbolic names instead (listing will display these names as well as addresses).</p> <p>Be careful when embedding assembly code - BASIC will not check if assembly instruction changed memory locations already used by BASIC variables.</p>
Example	<pre>asm movlw 67 movwf TMR0 end asm</pre>

PIC, PICmicro, and MPLAB are registered and protected trademarks of the Microchip Technology Inc. USA. Microchip logo and name are the registered tokens of the Microchip Technology. mikroBasic is a registered trade mark of mikroElektronika. All other tokens mentioned in the book are the property of the companies to which they belong.

mikroElektronika © 1998 - 2004. All rights reserved. If you have any questions, please contact our **office**.

Chapter 5: Built-in and Library Routines

- Introduction

- 5.1 Built-in Routines

- 5.1.1 SetBit
- 5.1.2 ClearBit
- 5.1.3 TestBit
- 5.1.4 Lo
- 5.1.5 Hi
- 5.1.6 Higher
- 5.1.7 Highest
- 5.1.8 Delay_us
- 5.1.9 Delay_ms
- 5.1.10 Inc
- 5.1.11 Dec
- 5.1.12 Length

- 5.2.6 EEPROM Library

- 5.2.6.1 EEPROM_Read
- 5.2.6.2 EEPROM_Write

- 5.2.7 Flash Memory Library

- 5.2.7.1 Flash_Read
- 5.2.7.2 Flash_Write

- 5.2.8 I2C Library

- 5.2.8.1 I2C_Init
- 5.2.8.2 I2C_Start
- 5.2.8.3 I2C_Repeated_Start
- 5.2.8.4 I2C_Rd
- 5.2.8.5 I2C_Wr
- 5.2.8.6 I2C_Stop

- 5.2.9 LCD Library

- 5.2.13 PWM Library

- 5.2.13.1 PWM_Init
- 5.2.13.2 PWM_Change_Duty
- 5.2.13.3 PWM_Start
- 5.2.13.4 PWM_Stop

- 5.2.14 RS485 Library

- 5.2.14.1 RS485Master_Init
- 5.2.14.2 RS485Master_Read
- 5.2.14.3 RS485Master_Write
- 5.2.14.4 RS485Slave_Init
- 5.2.14.5 RS485Slave_Read
- 5.2.14.6 RS485Slave_Write

- 5.2.15 SPI Library

- 5.2.15.1 SPI_Init
- 5.2.15.2 SPI_Init_Advanced

- 5.2 Library Routines

- 5.2.1 Numeric Formatting

- 5.2.1.1 ByteToStr
- 5.2.1.2 WordToStr
- 5.2.1.3 ShortToStr
- 5.2.1.4 IntToStr
- 5.2.1.5 Bcd2Dec
- 5.2.1.6 Dec2Bcd
- 5.2.1.7 Bcd2Dec16
- 5.2.1.8 Dec2Bcd16

- 5.2.2 ADC Library

- 5.2.2.1 ADC_read

- 5.2.3 CAN Library

- 5.2.3.1 CANSetOperationMode
- 5.2.3.2 CANGetOperationMode
- 5.2.3.3 CANInitialize
- 5.2.3.4 CANSetBaudRate
- 5.2.3.5 CANSetMask
- 5.2.3.6 CANSetFilter
- 5.2.3.7 CANWrite
- 5.2.3.8 CANRead
- 5.2.3.9 CAN Library Constants

- 5.2.9.1 LCD_Init

- 5.2.9.2 LCD_Config
- 5.2.9.3 LCD_Chr
- 5.2.9.4 LCD_Chr_CP
- 5.2.9.5 LCD_Out
- 5.2.9.6 LCD_Out_CP
- 5.2.9.7 LCD_Cmd

- 5.2.10 LCD8 Library

- 5.2.10.1 LCD8_Init
- 5.2.10.2 LCD8_Config
- 5.2.10.3 LCD8_Chr
- 5.2.10.4 LCD8_Chr_CP
- 5.2.10.5 LCD8_Out
- 5.2.10.6 LCD8_Out_CP
- 5.2.10.7 LCD8_Cmd

- 5.2.11 Graphic LCD Library

- 5.2.11.1 GLCD_Config
- 5.2.11.2 GLCD_Init
- 5.2.11.3 GLCD_Put_Ins
- 5.2.11.4 GLCD_Put_Data
- 5.2.11.5 GLCD_Put_Data2
- 5.2.11.6 GLCD_Select_Side
- 5.2.11.7 GLCD_Data_Read

- 5.2.15.3 SPI_Read

- 5.2.15.4 SPI_Write

- 5.2.16 USART Library

- 5.2.16.1 USART_Init
- 5.2.16.2 USART_Data_Ready
- 5.2.16.3 USART_Read
- 5.2.16.4 USART_Write

- 5.2.17 One-wire Library

- 5.2.17.1 OW_Reset
- 5.2.17.2 OW_Read
- 5.2.17.3 OW_Write

- 5.2.18 Software I2C Library

- 5.2.18.1 Soft_I2C_Config
- 5.2.18.2 Soft_I2C_Start
- 5.2.18.3 Soft_I2C_Write
- 5.2.18.4 Soft_I2C_Read
- 5.2.18.5 Soft_I2C_Stop

- 5.2.19 Software SPI Library

- 5.2.19.1 Soft_SPI_Config
- 5.2.19.2 Soft_SPI_Read
- 5.2.19.3 Soft_SPI_Write

- 5.2.4 CANSPI Library
- 5.2.4.1 CANSPISetOperationMode
- 5.2.4.2 CANSPIGetOperationMode
- 5.2.4.3 CANSPIInitialize
- 5.2.4.4 CANSPISetBaudRate
- 5.2.4.5 CANSPISetMask
- 5.2.4.6 CANSPISetFilter
- 5.2.4.7 CANSPIWrite
- 5.2.4.8 CANSPIRead
- 5.2.4.9 CANSPI Library Constants
- 5.2.5 Compact Flash Library
- 5.2.5.1 CF_Init_Port
- 5.2.5.2 CF_Detect
- 5.2.5.3 CF_Write_Init
- 5.2.5.4 CF_Write_Byte
- 5.2.5.5 CF_Write_Word
- 5.2.5.6 CF_Read_Init
- 5.2.5.7 CF_Read_Byte
- 5.2.5.8 CF_Read_Word
- 5.2.5.9 CF_File_Write_Init
- 5.2.5.10 CF_File_Write_Byte
- 5.2.5.11 CF_File_Write_Complete

- 5.2.11.8 GLCD_Clear_Dot
- 5.2.11.9 GLCD_Set_Dot
- 5.2.11.10 GLCD_Circle
- 5.2.11.11 GLCD_Line
- 5.2.11.12 GLCD_Invert
- 5.2.11.13 GLCD_Goto_XY
- 5.2.11.14 GLCD_Put_Char
- 5.2.11.15 GLCD_Clear_Screen
- 5.2.11.16 GLCD_Put_Text
- 5.2.11.17 GLCD_Rectangle
- 5.2.11.18 GLCD_Set_Font
- 5.2.12 Manchester Code Library
- 5.2.12.1 Man_Receive_Init
- 5.2.12.2 Man_Receive_Config
- 5.2.12.3 Man_Receive
- 5.2.12.4 Man_Send_Init
- 5.2.12.5 Man_Send_Config
- 5.2.12.6 Man_Send

- 5.2.20 Software UART Library
- 5.2.20.1 Soft_UART_Init
- 5.2.20.2 Soft_UART_Read
- 5.2.20.3 Soft_UART_Write
- 5.2.21 Sound Library
- 5.2.21.1 Sound_Init
- 5.2.21.2 Sound_Play
- 5.2.22 Trigonometry Library
- 5.2.22.1 SinE3
- 5.2.22.2 CosE3
- 5.2.23 Utilities
- 5.2.23.1 Button

Introduction

BASIC was designed with focus on simplicity of use. Great number of built-in and library routines are included to help you develop your applications quickly and easily.

5.1 Built-in Routines

BASIC incorporates a set of built-in functions and procedures. They are provided to make writing programs faster and easier. You can call built-in functions and procedures in any part of the program.

5.1.1 SetBit – *Sets the specified bit*

Prototype	<code>sub procedure SetBit(dim byref Reg as byte, dim Bit as byte)</code>
Description	Sets <Bit> of register <Reg>. Any SFR (Special Function Register) or variable of byte type can pass as valid variable parameter, but constants should be in range [0..7].
Example	<code>SetBit(PORTB,2) ' set bit RB2</code>

5.1.2 ClearBit – Clears the specified bit

Prototype	sub procedure ClearBit(dim byref Reg as byte , dim Bit as byte)
Description	Clears <Bit> of register <Reg>. Any SFR (Special Function Register) or variable of byte type can pass as valid variable parameter, but constants should be in range [0..7].
Example	ClearBit(PORTC,7) <i>' clear bit RC7</i>

5.1.3 TestBit – Tests the specified bit

Prototype	sub function TestBit(dim byref Reg as byte , dim Bit as byte) as byte
Description	Tests <Bit> of register <Reg>. If set, returns 1, otherwise 0. Any SFR (Special Function Register) or variable of byte type can pass as valid variable parameter, but constants should be in range [0..7].
Example	TestBit(PORTA,2) <i>' returns 1 if PORTA bit RA2 is 1, returns 0 otherwise</i>

5.1.4 Lo – Extract one byte from the specified parameter

Prototype	sub function Lo(dim Par as byte.. longint) as byte
Description	Returns byte 0 of <Par>, assuming that word/integer comprises bytes 1 and 0, and longint comprises bytes 3, 2, 1, and 0.
Example	Lo(A) ' returns lower byte of variable A

5.1.5 Hi – Extract one byte from the specified parameter

Prototype	sub function Hi(dim arg as word.. longint) as byte
Description	Returns byte 1 of <Par>, assuming that word/integer comprises bytes 1 and 0, and longint comprises bytes 3, 2, 1, and 0.
Example	Hi(Aa) ' returns hi byte of variable Aa

5.1.6 Higher – Extract one byte from the specified parameter

Prototype	sub function Higher(dim Par as longint) as byte
Description	Returns byte 2 of <Par>, assuming that longint comprises bytes 3, 2, 1, and 0.

Example	<code>Higher(Aaaa) ' returns byte next to the highest byte of variable Aaaa</code>
----------------	--

5.1.7 Highest – Extract one byte from the specified parameter

Prototype	<code>sub function Highest(dim arg as longint) as byte</code>
Description	Returns byte 3 of <i><Par></i> , assuming that longint comprises bytes 3, 2, 1, and 0.
Example	<code>Highest(Aaaa) ' returns the highest byte of variable Aaaa</code>

5.1.8 Delay_us – Software delay in us

Prototype	<code>sub procedure Delay_us(const Count as word)</code>
Description	Routine creates a software delay in duration of <i><Count></i> microseconds.
Example	<code>Delay_us(100) ' creates software delay equal to 1s</code>

5.1.9 Delay_ms – Software delay in ms

Prototype	sub procedure Delay_ms(const Count as word)
Description	Routine creates a software delay in duration of <i><Count></i> milliseconds.
Example	Delay_ms(1000) <i>' creates software delay equal to 1s</i>

5.1.10 Inc – *Increases variable by 1*

Prototype	sub procedure Inc(byref Par as byte..longint)
Description	Routine increases <i><Par></i> by one.
Example	Inc(Aaaa) <i>' increments variable Aaaa by 1</i>

5.1.11 Dec – *Decreases variable by 1*

Prototype	sub procedure Dec(byref Par as byte..longint)
Description	Routine decreases <i><Par></i> by one.

Example	<code>Dec(Aaaa) ' decrements variable Aaaa by 1</code>
----------------	--

5.1.12 Length – Returns length of string

Prototype	<code>sub function Length(dim Text as string) as byte</code>
Description	Routine returns length of string <i><Text></i> as byte.
Example	<code>Length(Text) ' returns string length as byte</code>

5.2 Library Routines

A comprehensive collection of functions and procedures is provided for simplifying the initialization and use of PIC MCU and its hardware modules. Routines currently includes libraries for ADC, I2C, USART, SPI, PWM, driver for LCD, drivers for internal and external CAN modules, flexible 485 protocol, numeric formatting routines...

5.2.1 Numeric Formatting Routines

Numeric formatting routines convert byte, short, word, and integer to string. You can get text representation of numerical value by passing it to one of the routines listed below.

5.2.1.1 ByteToStr – Converts byte to string

Prototype	sub procedure ByteToStr(dim input as byte, dim byref txt as char[6])
Description	<p>Parameter <i><input></i> represents numerical value of byte type that should be converted to string; parameter <i><txt></i> is passed by the address and contains the result of conversion.</p> <p>Parameter <i><txt></i> has to be of sufficient size to fit the converted string.</p>
Example	<pre>ByteToStr(Counter, Message) ' Copies value of byte Counter into string Message</pre>

5.2.1.2 WordToStr – Converts word to string

Prototype	sub procedure WordToStr(dim input as word, dim byref txt as char[6])
Description	<p>Parameter <i><input></i> represents numerical value of word type that should be converted to string; parameter <i><txt></i> is passed by the address and contains the result of conversion.</p> <p>Parameter <i><txt></i> has to be of sufficient size to fit the converted string.</p>

Example	<pre>WordToStr(Counter, Message) ' Copies value of word Counter into string Message</pre>
----------------	---

5.2.1.3 ShortToStr – Converts short to string

Prototype	sub procedure ShortToStr(dim input as short , dim byref txt as char [6])
Description	<p>Parameter <i><input></i> represents numerical value of short type that should be converted to string; parameter <i><txt></i> is passed by the address and contains the result of conversion.</p> <p>Parameter <i><txt></i> has to be of sufficient size to fit the converted string.</p>
Example	<pre>ShortToStr(Counter, Message) ' Copies value of short Counter into string Message</pre>

5.2.1.4 IntToStr – Converts integer to string

Prototype	sub procedure IntToStr(dim input as integer , dim byref txt as char [6])

Description	<p>Parameter <i><input></i> represents numerical value of integer type that should be converted to string; parameter <i><txt></i> is passed by the address and contains the result of conversion.</p> <p>Parameter <i><txt></i> has to be of sufficient size to fit the converted string.</p>
Example	<pre>IntToStr(Counter, Message) ' Copies value of integer Counter into string Message</pre>

5.2.1.5 Bcd2Dec – Converts 8-bit BCD value to decimal

Prototype	sub procedure Bcd2Dec(dim bcd_num as byte) as byte
Description	Function converts 8-bit BCD numeral to its decimal equivalent and returns the result as byte.
Example	<pre>dim a as byte dim b as byte ... a = 140 b = Bcd2Dec(a) ' b equals 224 now</pre>

5.2.1.6 Bcd2Dec – Converts 8-bit decimal to BCD

Prototype	sub procedure Dec2Bcd(dim dec_num as byte) as byte
Description	Function converts 8-bit decimal numeral to BCD and returns the result as byte.
Example	<pre> dim a as byte dim b as byte ... a = 224 b = Dec2Bcd(a) ' b equals 140 now </pre>

5.2.1.7 Bcd2Dec – Converts 16-bit BCD value to decimal

Prototype	sub procedure Bcd2Dec16(dim bcd_num as word) as word
Description	Function converts 16-bit BCD numeral to its decimal equivalent and returns the result as byte.
Example	<pre> dim a as word dim b as word ... a = 1234 b = Bcd2Dec16(a) ' b equals 4660 now </pre>

5.2.1.8 Bcd2Dec – Converts 16-bit BCD value to decimal

Prototype	sub procedure Dec2Bcd16(dim dec_num as word) as word
Description	Function converts 16-bit decimal numeral to BCD and returns the result as word.
Example	<pre> dim a as word dim b as word ... a = 4660 b = Dec2Bcd16(a) ' b equals 1234 now </pre>

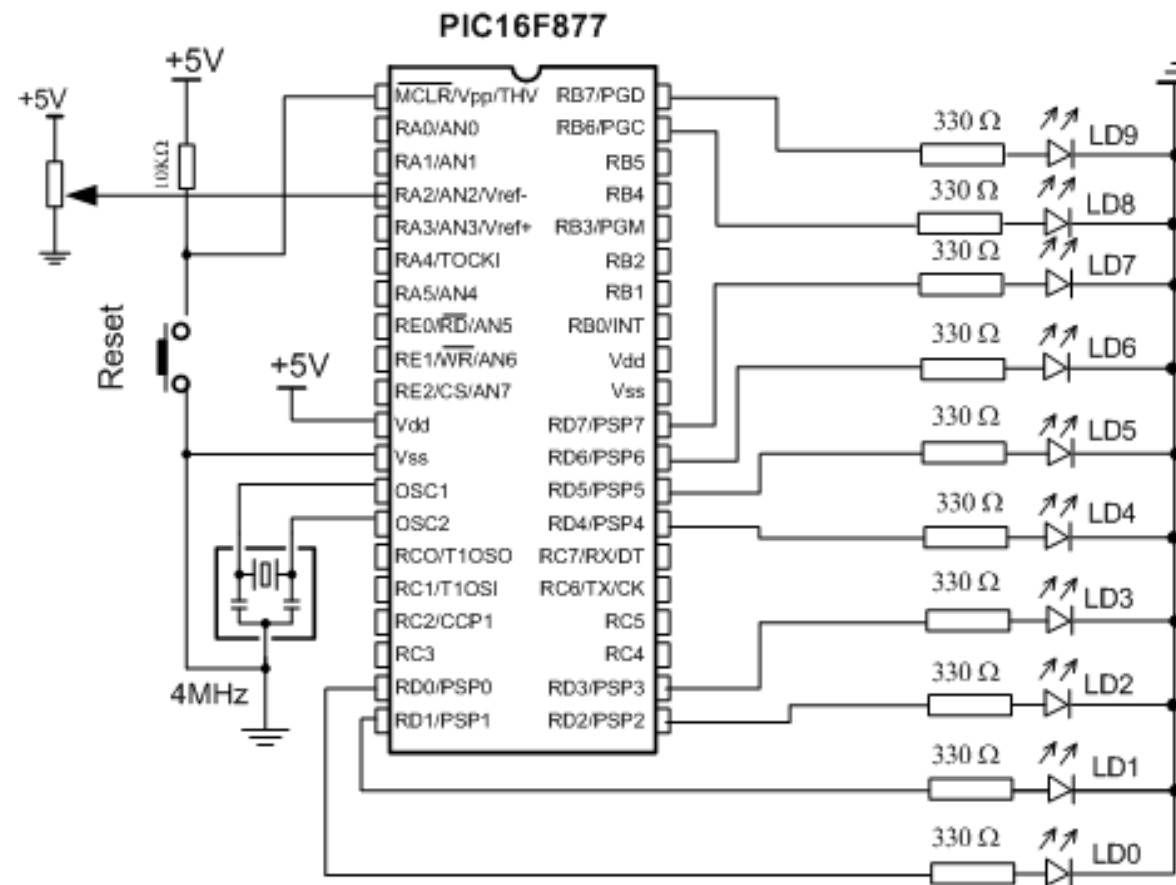
5.2.2 ADC Library

ADC (Analog to Digital Converter) module is available with a number of PIC MCU models. Library function ADC_Read is included to provide you comfortable work with the module. The function is currently unsupported by the following PIC MCU models: P18F2331, P18F2431, P18F4331, and P18F4431.

5.2.2.1 ADC_Read – Get the results of AD conversion

Prototype	sub function ADC_Read(dim Channel as byte) as word
------------------	--

Description	Routine initializes ADC module to work with RC clock. Clock determines the time period necessary for performing AD conversion (min 12TAD). RC sources typically have T_{ad} 4 μ S. Parameter <i><Channel></i> determines which channel will be sampled. Refer to the device data sheet for information on device channels.
Example	<code>res = ADC_Read(2)</code> ' reads channel 2 and stores value in variable res



ADC HW connection

5.2.3 CAN Library

The Controller Area Network module (CAN) is serial interface, used for communicating with other peripherals or microcontrollers. CAN module is available with a number of PIC MCU models. BASIC includes a set of library routines to provide you comfortable work with the module. More details about CAN can be found in appropriate literature and on mikroElektronika Web site.

5.2.3.1 CANSetOperationMode – Sets CAN to requested mode

Prototype	sub procedure CANSetOperationMode(dim Mode as byte , dim Wait as byte)
Description	<p>The procedure copies <i><Mode></i> to CANSTAT and sets CAN to requested mode.</p> <p>Operation <i><Mode></i> code can take any of predefined constant values. <i><Wait></i> takes values TRUE(255) or FALSE(0)</p> <p>If Wait is true, this is a blocking call. It won't return until requested mode is set. If Wait is false, this is a non-blocking call. It does not verify if CAN module is switched to requested mode or not. Caller must use CANGetOperationMode() to verify correct operation mode before performing mode specific operation.</p>
Example	CANSetOperationMode(CAN_MODE_LISTEN, TRUE) <i>' Sets CAN to Listen mode</i>

5.2.3.2 CANGetOperationMode – Returns the current operation mode of CAN

Prototype	sub function CANGetOperationMode as byte
Description	The function returns the current operation mode of CAN.
Example	CANGetOperationMode

5.2.3.3 CANInitialize – *Initializes CAN*

Prototype	sub procedure CANInitialize(dim SJW as byte , dim BRP as byte , dim PHSEG1 as byte , dim PHSEG2 as byte , dim PROPSEG as byte , dim CAN_CONFIG_FLAGS as byte)
Description	<p>The procedure initializes CAN module. CAN must be in Configuration mode or else these values will be ignored.</p> <p>Parameters:</p> <p>SJW value as defined in 18XXX8 datasheet (must be between 1 thru 4)</p> <p>BRP value as defined in 18XXX8 datasheet (must be between 1 thru 64)</p> <p>PHSEG1 value as defined in 18XXX8 datasheet (must be between 1 thru 8)</p> <p>PHSEG2 value as defined in 18XXX8 datasheet (must be between 1 thru 8)</p> <p>PROPSEG value as defined in 18XXX8 datasheet (must be between 1 thru 8)</p> <p>CAN_CONFIG_FLAGS value is formed from constants (see below)</p> <p>Output:</p>

CAN bit rate is set. All masks registers are set to '0' to allow all messages.

Filter registers are set according to flag value:

```
If (CAN_CONFIG_FLAGS and CAN_CONFIG_VALID_XTD_MSG) <> 0
    Set all filters to XTD_MSG
Else if (config and CONFIG_VALID_STD_MSG) <> 0
    Set all filters to STD_MSG
Else
    Set half of the filters to STD, and the rest to XTD_MSG
```

Side Effects:

All pending transmissions are aborted.

Example

```
dim aa as byte
```

```
aa =    CAN_CONFIG_SAMPLE_THRICE and      ' form value to be used
        CAN_CONFIG_PHSEG2_PRG_ON and      ' with CANInitialize
        CAN_CONFIG_STD_MSG and
        CAN_CONFIG_DBL_BUFFER_ON and
        CAN_CONFIG_VALID_XTD_MSG and
        CAN_CONFIG_LINE_FILTER_OFF
```

```
CANInitialize(1, 1, 3, 3, 1, aa)
```

5.2.3.4 CANSetBaudRate – Sets CAN Baud Rate

Prototype	sub procedure CANSetBaudRate(dim SJW as byte , dim BRP as byte , dim PHSEG1 as byte , dim PHSEG2 as byte , dim PROPSEG as byte , dim CAN_CONFIG_FLAGS as byte)
Description	<p>The procedure sets CAN Baud Rate. CAN must be in Configuration mode or else these values will be ignored.</p> <p>Parameters:</p> <p>SJW value as defined in 18XXX8 datasheet (must be between 1 thru 4)</p> <p>BRP value as defined in 18XXX8 datasheet (must be between 1 thru 64)</p> <p>PHSEG1 value as defined in 18XXX8 datasheet (must be between 1 thru 8)</p> <p>PHSEG2 value as defined in 18XXX8 datasheet (must be between 1 thru 8)</p> <p>PROPSEG value as defined in 18XXX8 datasheet (must be between 1 thru 8)</p> <p>CAN_CONFIG_FLAGS - Value formed from constants (see section below)</p> <p>Output:</p> <p>Given values are bit adjusted to fit in 18XXX8 and BRGCONx registers and copied. CAN bit rate is set as per given values.</p>
Example	CANSetBaudRate(1, 1, 3, 3, 1, aa)

5.2.3.5 CANSetMask – Sets the CAN message mask

Prototype	sub procedure CANSetMask(CAN_MASK as byte , val as longint , dim CAN_CONFIG_FLAGS as byte)
Description	<p>The procedure sets the CAN message mask. CAN must be in Configuration mode. If not, all values will be ignored.</p> <p>Parameters: CAN_MASK - One of predefined constant value val - Actual mask register value CAN_CONFIG_FLAGS - Type of message to filter, either CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG</p> <p>Output: Given value is bit adjusted to appropriate buffer mask registers.</p>
Example	CANSetMask(CAN_MASK_B2, -1, CAN_CONFIG_XTD_MSG)

5.2.3.6 CANSetFilter – Sets the CAN message filter

Prototype	sub procedure CANSetFilter(dim CAN_FILTER as byte , dim val as longint , dim CAN_CONFIG_FLAGS as byte)

Description	<p>The procedure sets the CAN message filter. CAN must be in Configuration mode. If not, all values will be ignored.</p> <p>Parameters:</p> <p>CAN_FILTER - One of predefined constant values</p> <p>val - Actual filter register value.</p> <p>CAN_CONFIG_FLAGS - Type of message to filter, either CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG</p> <p>Output:</p> <p>Given value is bit adjusted to appropriate buffer filter registers</p>
Example	<pre>CANSetFilter(CAN_FILTER_B1_F1, 3, CAN_CONFIG_XTD_MSG)</pre>

5.2.3.7 CANWrite – *Queues message for transmission*

Prototype	<pre>sub function CANWrite(dim id as longint, dim byref Data : as byte[8], dim DataLen as byte, dim CAN_TX_MSG_FLAGS as byte) as byte</pre>
------------------	---

Description	<p>If at least one empty transmit buffer is found, given message is queued for the transmission. If none found, FALSE value is returned. CAN must be in Normal mode.</p> <p>Parameters:</p> <p>id - CAN message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended)</p> <p>Data - array of bytes up to 8 bytes in length</p> <p>DataLen - Data length from 1 thru 8</p> <p>CAN_TX_MSG_FLAGS - Value formed from constants (see section below)</p>
Example	<pre> a1 = CAN_TX_PRIORITY_0 and ' form value to be used CAN_TX_XTD_FRAME and ' with CANWrite CAN_TX_NO_RTR_FRAME CANWrite(-1, data, 1, a1) </pre>

5.2.3.8 CANRead – *Extracts and reads the message*

Prototype	<pre> sub function CANRead(dim byref id as longint, dim byref Data as byte[8], dim byref DataLen as byte, dim byref CAN_RX_MSG_FLAGS as byte) as byte </pre>
------------------	--

Description	<p>If at least one full receive buffer is found, the function extracts and returns the message as byte. If none found, FALSE value is returned. CAN must be in mode in which receiving is possible.</p> <p>Parameters:</p> <p>id - CAN message identifier</p> <p>Data - array of bytes up to 8 bytes in length</p> <p>DataLen - Data length from 1 thru 8</p> <p>CAN_TX_MSG_FLAGS - Value formed from constants (see below)</p>
Example	<pre>res = CANRead(id, Data, 7, 0)</pre>

5.2.3.9 CAN Library Constants

You need to be familiar with constants that are provided for use with the CAN module. All of the following constants are predefined in CAN library.

CAN_OP_MODE

These constant values define CAN module operation mode. CANSetOperationMode() routine requires this code. These values must be used by itself, i.e. they cannot be ANDed to form multiple values.

```

const CAN_MODE_BITS      = $E0    ' Use these to access opmode bits
const CAN_MODE_NORMAL    = 0
const CAN_MODE_SLEEP      = $20

```

```

const CAN_MODE_LOOP      = $40
const CAN_MODE_LISTEN    = $60
const CAN_MODE_CONFIG    = $80

```

CAN_TX_MSG_FLAGS

These constant values define flags related to transmission of a CAN message. There could be more than one this flag ANDed together to form multiple flags.

```

const CAN_TX_PRIORITY_BITS = $03

const CAN_TX_PRIORITY_0    = $FC          ' XXXXXX00
const CAN_TX_PRIORITY_1    = $FD          ' XXXXXX01
const CAN_TX_PRIORITY_2    = $FE          ' XXXXXX10
const CAN_TX_PRIORITY_3    = $FF          ' XXXXXX11

const CAN_TX_FRAME_BIT     = $08

const CAN_TX_STD_FRAME     = $FF          ' XXXXX1XX
const CAN_TX_XTD_FRAME     = $F7          ' XXXXX0XX

const CAN_TX_RTR_BIT       = $40

const CAN_TX_NO_RTR_FRAME  = $FF          ' X1XXXXXX
const CAN_TX_RTR_FRAME     = $BF          ' X0XXXXXX

```

CAN_RX_MSG_FLAGS

These constant values define flags related to reception of a CAN message. There could be more than one this flag ANDed together to form multiple flags. If a particular bit is set; corresponding meaning is TRUE or else it will be FALSE.

e.g.

```
if (MsgFlag and CAN_RX_OVERFLOW) <> 0 then
```

```
    ' Receiver overflow has occurred.
```

```
    ' We have lost our previous message.
```

```
const CAN_RX_FILTER_BITS = $07    ' Use these to access filter bits
```

```
const CAN_RX_FILTER_1 = $00
```

```
const CAN_RX_FILTER_2 = $01
```

```
const CAN_RX_FILTER_3 = $02
```

```
const CAN_RX_FILTER_4 = $03
```

```
const CAN_RX_FILTER_5 = $04
```

```
const CAN_RX_FILTER_6 = $05
```

```
const CAN_RX_OVERFLOW = $08        ' Set if Overflowed else cleared
```

```
const CAN_RX_INVALID_MSG = $10     ' Set if invalid else cleared
```

```
const CAN_RX_XTD_FRAME = $20       ' Set if XTD message else cleared
```

```
const CAN_RX_RTR_FRAME = $40       ' Set if RTR message else cleared
```

```
const CAN_RX_DBL_BUFFERED = $80    ' Set if this message was hardware double-buffered
```

CAN_MASK

These constant values define mask codes. Routine CANSetMask() requires this code as one of its arguments. These enumerations must be used by itself i.e. it cannot be ANDed to form multiple values.

```
const CAN_MASK_B1 = 0
const CAN_MASK_B2 = 1
```

CAN_FILTER

These constant values define filter codes. Routine CANSetFilter() requires this code as one of its arguments. These enumerations must be used by itself, i.e. it cannot be ANDed to form multiple values.

```
const CAN_FILTER_B1_F1 = 0
const CAN_FILTER_B1_F2 = 1
const CAN_FILTER_B2_F1 = 2
const CAN_FILTER_B2_F2 = 3
const CAN_FILTER_B2_F3 = 4
const CAN_FILTER_B2_F4 = 5
```

CAN_CONFIG_FLAGS

These constant values define flags related to configuring CAN module. Routines CANInitialize() and CANSetBaudRate() use these codes. One or more these values may be ANDed to form multiple flags

```
const CAN_CONFIG_DEFAULT = $FF          ' 11111111

const CAN_CONFIG_PHSEG2_PRG_BIT = $01
```



```

const CAN_CONFIG_PHSEG2_PRG_ON = $FF      ' XXXXXXXX1
const CAN_CONFIG_PHSEG2_PRG_OFF = $FE     ' XXXXXXXX0

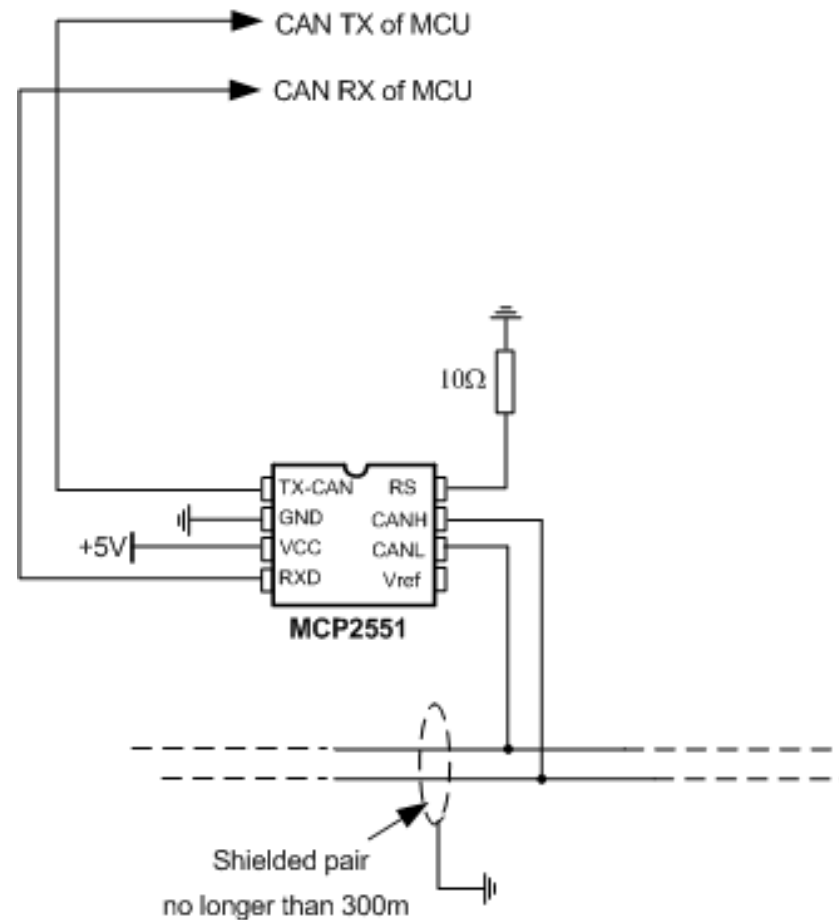
const CAN_CONFIG_LINE_FILTER_BIT = $02
const CAN_CONFIG_LINE_FILTER_ON = $FF      ' XXXXXXX1X
const CAN_CONFIG_LINE_FILTER_OFF = $FD    ' XXXXXXX0X

const CAN_CONFIG_SAMPLE_BIT = $04
const CAN_CONFIG_SAMPLE_ONCE = $FF        ' XXXXX1XX
const CAN_CONFIG_SAMPLE_THRICE = $FB     ' XXXXX0XX

const CAN_CONFIG_MSG_TYPE_BIT = $08
const CAN_CONFIG_STD_MSG = $FF            ' XXXX1XXX
const CAN_CONFIG_XTD_MSG = $F7           ' XXXX0XXX

const CAN_CONFIG_DBL_BUFFER_BIT = $10
const CAN_CONFIG_DBL_BUFFER_ON = $FF      ' XXX1XXXX
const CAN_CONFIG_DBL_BUFFER_OFF = $EF    ' XXX0XXXX

const CAN_CONFIG_MSG_BITS = $60
const CAN_CONFIG_ALL_MSG = $FF           ' X11XXXXX
const CAN_CONFIG_VALID_XTD_MSG = $DF     ' X10XXXXX
const CAN_CONFIG_VALID_STD_MSG = $BF     ' X01XXXXX
const CAN_CONFIG_ALL_VALID_MSG = $9F     ' X00XXXXX
    
```



Example of interfacing CAN transceiver with MCU and bus

5.2.4 CANSPI Library

The Controller Area Network module (CAN) is serial interface, used for communicating with other peripherals or microcontrollers. CAN module is available with a number of PIC MCU models. MCP2515 or MCP2510 are modules that enable any chip with SPI interface to communicate over CAN bus. BASIC includes a set of library routines to provide you comfortable work with the

module. More details about CAN can be found in appropriate literature and on mikroElektronika Web site.

Note: CANSPI routines are supported by any PIC MCU model that has SPI interface on PORTC. Also, CS pin of MCP2510 or MCP2515 must be connected to RC0 pin.

5.2.4.1 CANSPISetOperationMode – Sets CAN to requested mode

Prototype	sub procedure CANSPISetOperationMode(dim mode as byte , dim Wait as byte)
Description	<p>The procedure copies <i><mode></i> to CANSTAT and sets CAN to requested mode.</p> <p>Operation <i><mode></i> code can take any of predefined constant values. <i><Wait></i> takes values TRUE(255) or FALSE(0)</p> <p>If Wait is true, this is a blocking call. It won't return until requested mode is set. If Wait is false, this is a non-blocking call. It does not verify if CAN module is switched to requested mode or not. Caller must use CANGetOperationMode() to verify correct operation mode before performing mode specific operation.</p>
Example	CANSPISetOperationMode(CAN_MODE_LISTEN, TRUE) <i>' Sets CAN to Listen mode</i>

5.2.4.2 CANSPIGetOperationMode – Returns the current operation mode of CAN

Prototype	sub function CANSPIGetOperationMode as byte
------------------	---

Description	The function returns the current operation mode of CAN.
Example	CANGetOperationMode

5.2.4.3 CANSPIInitialize – *Initializes CANSPI*

Prototype	<pre>sub procedure CANSPIInitialize(dim SJW as byte, dim BRP as byte, dim PHSEG1 as byte, dim PHSEG2 as byte, dim PROPSEG as byte, dim CAN_CONFIG_FLAGS as byte)</pre>
Description	<p>The procedure initializes CAN module. CAN must be in Configuration mode or else these values will be ignored.</p> <p>Parameters:</p> <p>SJW value as defined in 18XXX8 datasheet (must be between 1 thru 4)</p> <p>BRP value as defined in 18XXX8 datasheet (must be between 1 thru 64)</p> <p>PHSEG1 value as defined in 18XXX8 datasheet (must be between 1 thru 8)</p> <p>PHSEG2 value as defined in 18XXX8 datasheet (must be between 1 thru 8)</p> <p>PROPSEG value as defined in 18XXX8 datasheet (must be between 1 thru 8)</p> <p>CAN_CONFIG_FLAGS value is formed from constants (see below)</p> <p>Output:</p> <p>CAN bit rate is set. All masks registers are set to '0' to allow all messages.</p> <p>Filter registers are set according to flag value:</p>

```

If (CAN_CONFIG_FLAGS and CAN_CONFIG_VALID_XTD_MSG) <> 0
    Set all filters to XTD_MSG
Else if (config and CONFIG_VALID_STD_MSG) <> 0
    Set all filters to STD_MSG
Else
    Set half of the filters to STD, and the rest to XTD_MSG

```

Side Effects:

All pending transmissions are aborted.

Example

```
dim aa as byte
```

```

aa =    CAN_CONFIG_SAMPLE_THRICE and      ' form value to be used
        CAN_CONFIG_PHSEG2_PRG_ON and      ' with CANSPIInitialize
        CAN_CONFIG_STD_MSG and
        CAN_CONFIG_DBL_BUFFER_ON and
        CAN_CONFIG_VALID_XTD_MSG and
        CAN_CONFIG_LINE_FILTER_OFF

```

```
CANInitialize(1, 1, 3, 3, 1, aa)
```

5.2.4.4 CANSPISetBaudRate – Sets CAN Baud Rate

Prototype	sub procedure CANSPISetBaudRate(dim SJW as byte , dim BRP as byte , dim PHSEG1 as byte , dim PHSEG2 as byte , dim PROPSEG as byte , dim CAN_CONFIG_FLAGS as byte)
Description	<p>The procedure sets CAN Baud Rate. CAN must be in Configuration mode or else these values will be ignored.</p> <p>Parameters:</p> <p>SJW value as defined in 18XXX8 datasheet (must be between 1 thru 4)</p> <p>BRP value as defined in 18XXX8 datasheet (must be between 1 thru 64)</p> <p>PHSEG1 value as defined in 18XXX8 datasheet (must be between 1 thru 8)</p> <p>PHSEG2 value as defined in 18XXX8 datasheet (must be between 1 thru 8)</p> <p>PROPSEG value as defined in 18XXX8 datasheet (must be between 1 thru 8)</p> <p>CAN_CONFIG_FLAGS - Value formed from constants (see section below)</p> <p>Output:</p> <p>Given values are bit adjusted to fit in 18XXX8 and BRGCONx registers and copied. CAN bit rate is set as per given values.</p>
Example	CANSPISetBaudRate(1, 1, 3, 3, 1, aa)

5.2.4.5 CANSPISetMask – Sets the CAN message mask

Prototype	sub procedure CANSPISetMask(CAN_MASK as byte , val as longint , dim CAN_CONFIG_FLAGS as byte)
Description	<p>The procedure sets the CAN message mask. CAN must be in Configuration mode. If not, all values will be ignored.</p> <p>Parameters: CAN_MASK - One of predefined constant value val - Actual mask register value CAN_CONFIG_FLAGS - Type of message to filter, either CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG</p> <p>Output: Given value is bit adjusted to appropriate buffer mask registers.</p>
Example	CANSPISetMask(CAN_MASK_B2, -1, CAN_CONFIG_XTD_MSG)

5.2.4.6 CANSPISetFilter – *Sets the CAN message filter*

Prototype	sub procedure CANSPISetFilter(dim CAN_FILTER as byte , dim val as longint , dim CAN_CONFIG_FLAGS as byte)

Description	<p>The procedure sets the CAN message filter. CAN must be in Configuration mode. If not, all values will be ignored.</p> <p>Parameters:</p> <p>CAN_FILTER - One of predefined constant values</p> <p>val - Actual filter register value.</p> <p>CAN_CONFIG_FLAGS - Type of message to filter, either CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG</p> <p>Output:</p> <p>Given value is bit adjusted to appropriate buffer filter registers</p>
Example	<pre>CANSPISetFilter(CAN_FILTER_B1_F1, 3, CAN_CONFIG_XTD_MSG)</pre>

5.2.4.7 CANSPIWrite – *Queues message for transmission*

Prototype	<pre>sub function CANSPIWrite(dim id as longint, dim byref Data : as byte[8], dim DataLen as byte, dim CAN_TX_MSG_FLAGS as byte) as byte</pre>
------------------	--

Description	<p>If at least one empty transmit buffer is found, given message is queued for the transmission. If none found, FALSE value is returned. CAN must be in Normal mode.</p> <p>Parameters:</p> <p>id - CAN message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended)</p> <p>Data - array of as bytes up to 8 as bytes in length</p> <p>DataLen - Data length from 1 thru 8</p> <p>CAN_TX_MSG_FLAGS - Value formed from constants (see section below)</p>
Example	<pre> aa1 = CAN_TX_PRIORITY_0 and ' form value to be used CAN_TX_XTD_FRAME and ' with CANSPIWrite CAN_TX_NO_RTR_FRAME CANSPIWrite(-1, data, 1, aa1) </pre>

5.2.4.8 CANSPIRead – *Extracts and reads the message*

Prototype	<pre> sub function CANSPIRead(dim byref id as longint, dim byref Data as byte [8], dim byref DataLen as byte, dim byref CAN_RX_MSG_FLAGS as byte) as byte </pre>
------------------	--

Description	<p>If at least one full receive buffer is found, the function extracts and returns the message as byte. If none found, FALSE value is returned. CAN must be in mode in which receiving is possible.</p> <p>Parameters:</p> <p>id - CAN message identifier</p> <p>Data - array of bytes up to 8 bytes in length</p> <p>DataLen - Data length from 1 thru 8</p> <p>CAN_TX_MSG_FLAGS - Value formed from constants (see below)</p>
Example	<pre>res = CANSPIRead(id, Data, 7, 0)</pre>

5.2.4.9 CANSPI Library Constants

You need to be familiar with constants that are provided for use with the CAN module. All of the following constants are predefined in CANSPI library.

CAN_OP_MODE

These constant values define CAN module operation mode. CANSetOperationMode() routine requires this code. These values must be used by itself, i.e. they cannot be ANDed to form multiple values.

```

const CAN_MODE_BITS      = $E0    ' Use these to access opmode bits
const CAN_MODE_NORMAL    = 0
const CAN_MODE_SLEEP     = $20

```

```

const CAN_MODE_LOOP      = $40
const CAN_MODE_LISTEN    = $60
const CAN_MODE_CONFIG    = $80

```

CAN_TX_MSG_FLAGS

These constant values define flags related to transmission of a CAN message. There could be more than one this flag ANDed together to form multiple flags.

```

const CAN_TX_PRIORITY_BITS  = $03

const CAN_TX_PRIORITY_0     = $FC           ' XXXXXX00
const CAN_TX_PRIORITY_1     = $FD           ' XXXXXX01
const CAN_TX_PRIORITY_2     = $FE           ' XXXXXX10
const CAN_TX_PRIORITY_3     = $FF           ' XXXXXX11

const CAN_TX_FRAME_BIT     = $08

const CAN_TX_STD_FRAME      = $FF           ' XXXXX1XX
const CAN_TX_XTD_FRAME      = $F7           ' XXXXX0XX

const CAN_TX_RTR_BIT       = $40

const CAN_TX_NO_RTR_FRAME   = $FF           ' X1XXXXXX
const CAN_TX_RTR_FRAME     = $BF           ' X0XXXXXX

```

CAN_RX_MSG_FLAGS

These constant values define flags related to reception of a CAN message. There could be more than one this flag ANDed together to form multiple flags. If a particular bit is set; corresponding meaning is TRUE or else it will be FALSE.

e.g.

```
if (MsgFlag and CAN_RX_OVERFLOW) <> 0 then
```

```
    ' Receiver overflow has occurred.
```

```
    ' We have lost our previous message.
```

```
const CAN_RX_FILTER_BITS = $07    ' Use these to access filter bits
```

```
const CAN_RX_FILTER_1 = $00
```

```
const CAN_RX_FILTER_2 = $01
```

```
const CAN_RX_FILTER_3 = $02
```

```
const CAN_RX_FILTER_4 = $03
```

```
const CAN_RX_FILTER_5 = $04
```

```
const CAN_RX_FILTER_6 = $05
```

```
const CAN_RX_OVERFLOW = $08        ' Set if Overflowed else cleared
```

```
const CAN_RX_INVALID_MSG = $10     ' Set if invalid else cleared
```

```
const CAN_RX_XTD_FRAME = $20       ' Set if XTD message else cleared
```

```
const CAN_RX_RTR_FRAME = $40       ' Set if RTR message else cleared
```

```
const CAN_RX_DBL_BUFFERED = $80    ' Set if this message was hardware double-buffered
```

CAN_MASK

These constant values define mask codes. Routine CANSetMask() requires this code as one of its arguments. These enumerations must be used by itself i.e. it cannot be ANDed to form multiple values.

```
const CAN_MASK_B1 = 0
const CAN_MASK_B2 = 1
```

CAN_FILTER

These constant values define filter codes. Routine CANSetFilter() requires this code as one of its arguments. These enumerations must be used by itself, i.e. it cannot be ANDed to form multiple values.

```
const CAN_FILTER_B1_F1 = 0
const CAN_FILTER_B1_F2 = 1
const CAN_FILTER_B2_F1 = 2
const CAN_FILTER_B2_F2 = 3
const CAN_FILTER_B2_F3 = 4
const CAN_FILTER_B2_F4 = 5
```

CAN_CONFIG_FLAGS

These constant values define flags related to configuring CAN module. Routines CANInitialize() and CANSetBaudRate() use these codes. One or more these values may be ANDed to form multiple flags

```
const CAN_CONFIG_DEFAULT = $FF          ' 11111111

const CAN_CONFIG_PHSEG2_PRG_BIT = $01
```

```

const CAN_CONFIG_PHSEG2_PRG_ON = $FF      ' XXXXXXX1
const CAN_CONFIG_PHSEG2_PRG_OFF = $FE     ' XXXXXXX0

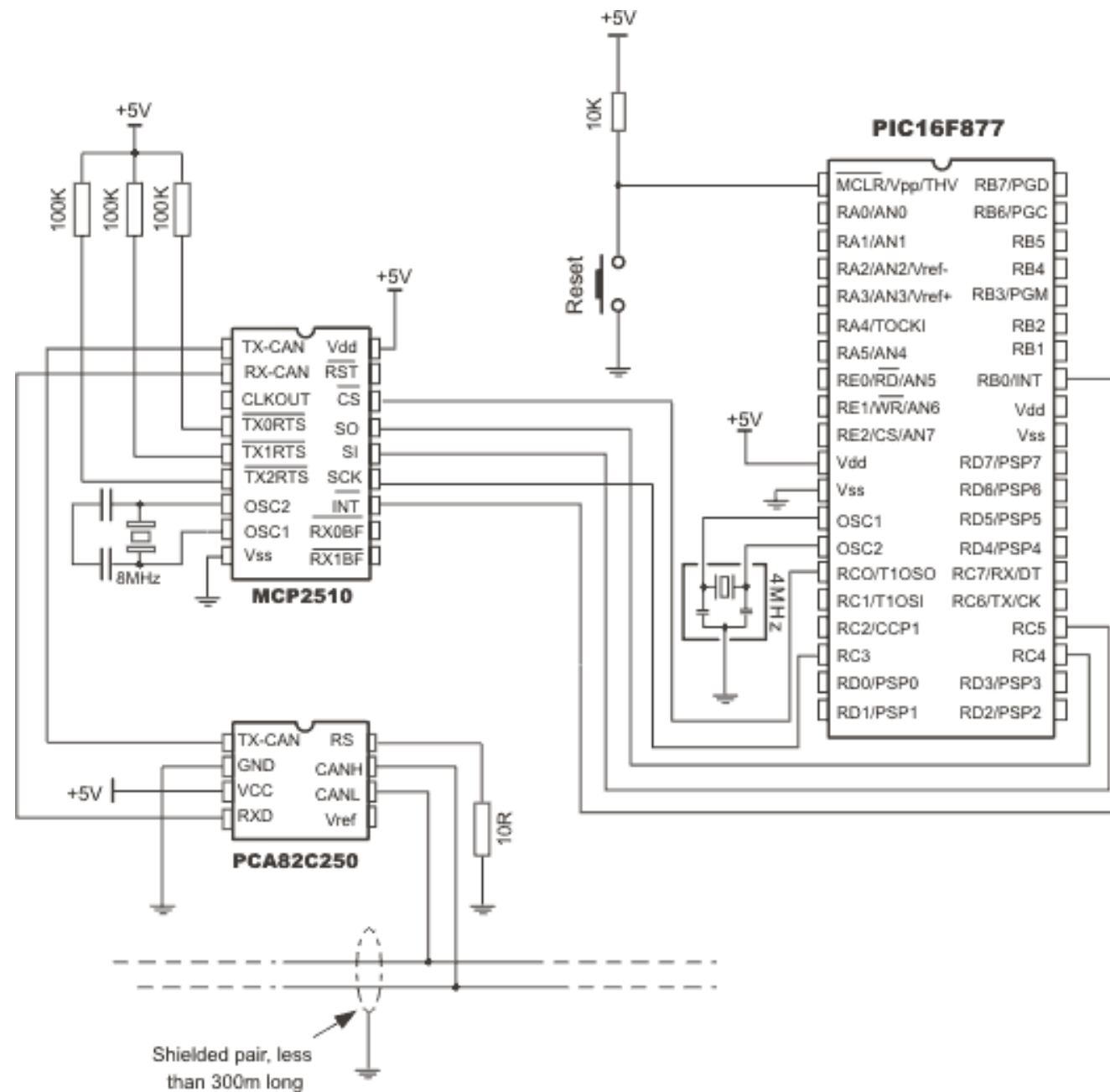
const CAN_CONFIG_LINE_FILTER_BIT = $02
const CAN_CONFIG_LINE_FILTER_ON = $FF     ' XXXXXX1X
const CAN_CONFIG_LINE_FILTER_OFF = $FD    ' XXXXXX0X

const CAN_CONFIG_SAMPLE_BIT = $04
const CAN_CONFIG_SAMPLE_ONCE = $FF        ' XXXXX1XX
const CAN_CONFIG_SAMPLE_THRICE = $FB     ' XXXXX0XX

const CAN_CONFIG_MSG_TYPE_BIT = $08
const CAN_CONFIG_STD_MSG = $FF            ' XXXX1XXX
const CAN_CONFIG_XTD_MSG = $F7           ' XXXX0XXX

const CAN_CONFIG_DBL_BUFFER_BIT = $10
const CAN_CONFIG_DBL_BUFFER_ON = $FF     ' XXX1XXXX
const CAN_CONFIG_DBL_BUFFER_OFF = $EF    ' XXX0XXXX

const CAN_CONFIG_MSG_BITS = $60
const CAN_CONFIG_ALL_MSG = $FF           ' X11XXXXX
const CAN_CONFIG_VALID_XTD_MSG = $DF     ' X10XXXXX
const CAN_CONFIG_VALID_STD_MSG = $BF     ' X01XXXXX
const CAN_CONFIG_ALL_VALID_MSG = $9F     ' X00XXXXX
    
```



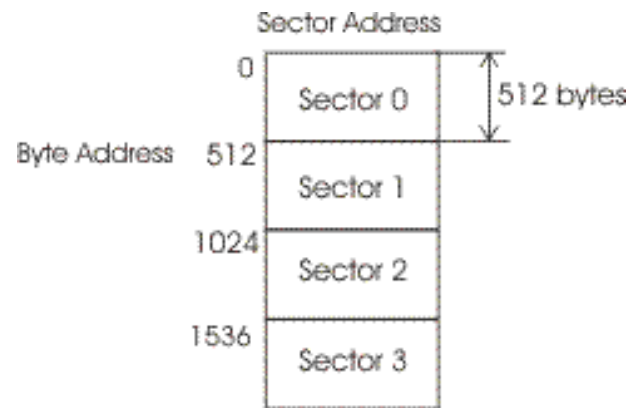
Example of interfacing CAN transceiver MCP2551, and MCP2510 with MCU and bus

5.2.5 Compact Flash Library

Compact Flash Library provides routines for accessing data on Compact Flash card (abbrev. CF further in text). CF cards are widely used memory elements, commonly found in digital cameras. Great capacity (8MB ~ 2GB, and more) and excellent access time of typically few microseconds make them very attractive for microcontroller applications.

In CF card, data is divided into sectors, one sector usually comprising 512 bytes (few older models have sectors of 256B). Read and write operations are not performed directly, but successively through 512B buffer. Following routines can be used for CF with FAT16 and FAT32 file system.

Note: routines for file handling (`CF_File_Write_Init`, `CF_File_Write_Byte`, `CF_File_Write_Complete`) can be used only with FAT16 file system, and only with PIC18 family!



Before write operation, make sure you don't overwrite boot or FAT sector as it could make your card on PC or digital cam unreadable. Drive mapping tools, such as Winhex, can be of a great assistance.

5.2.5.1 CF_Init_Port – *Initializes ports appropriately*

Prototype	sub procedure CF_INIT_PORT(dim byref CtrlPort as byte , dim byref DataPort as byte)
Description	The procedure initializes ports appropriately: <CtrlPort> is control port, and <DataPort> is data port to which CF is attached.
Example	CF_Init_Port(PORTB, PORTD) <i>' Control port is PORTB, Data port is PORTD</i>

5.2.5.2 CF_Detect – Checks for presence of CF

Prototype	sub function CF_DETECT(dim byref CtrlPort as byte) as byte
Description	The function checks if Compact Flash card is present. Returns true if present, otherwise returns false. <CtrlPort> must be initialized (call CF_INIT_PORT first).
Example	<pre> do nop loop until CF_Detect(PORTB) = true <i>' wait until CF card is inserted</i> </pre>

5.2.5.3 CF_Write_Init – Initializes CF card for writing

Prototype	sub procedure CF_WRITE_INIT(dim byref CtrlPort as byte , dim byref DataPort as byte , dim Adr as longint , dim SectCnt as byte)
Description	<p>The procedure initializes CF card for writing. Ports need to be initialized.</p> <p>Parameters:</p> <p>CtrlPort - control port, DataPort - data port, k - specifies sector address from where data will be written, SectCnt - parameter is total number of sectors prepared for write.</p>
Example	<pre>CF_Write_Init(PORTB, PORTD, 590, 1) ' Initialize write at sector address 590 ' of 1 sector (512 bytes)</pre>

5.2.5.4 CF_Write_Byte – Writes 1 byte to CF

Prototype	sub procedure CF_WRITE_BYTE(dim byref CtrlPort as byte , dim byref DataPort as byte , dim BData as byte)
------------------	---

Description	<p>The procedure writes 1 byte to Compact Flash. The procedure has effect only if CF card is initialized for writing.</p> <p>Parameters: CtrlPort - control port, DataPort - data port, dat - data byte written to CF</p>
Example	<pre>CF_Write_Init(PORTB, PORTD, 590, 1) ' Initialize write at sector address 590 ' of 1 sector (512 bytes) for i = 0 to 511 ' Write 512 bytes to sector at address 590 CF_Write_Byte(PORTB, PORTD, i) next i</pre>

5.2.5.5 CF_Write_Word – Writes 1 word to CF

Prototype	<pre>sub procedure CF_WRITE_WORD(dim byref CtrlPort as byte, dim byref DataPort as byte, dim WData as word)</pre>
------------------	---

Description	<p>The procedure writes 1 word to Compact Flash. The procedure has effect only if CF card is initialized for writing.</p> <p>Parameters: CtrlPort - control port, DataPort - data port, Wdata - data word written to CF</p>
Example	CF_Write_Word(PORTB, PORTD, Data)

5.2.5.6 CF_Read_Init – *Initializes CF card for reading*

Prototype	sub procedure CF_READ_INIT(dim byref CtrlPort as byte , dim byref DataPort as byte , dim Adr as longint , dim SectCnt as byte)
Description	<p>Parameters: CtrlPort - control port, DataPort - data port, Adr - specifies sector address from where data will be read, SectCnt - total number of sectors prepared for read operations.</p>

Example	<pre>CF_Read_Init(PORTB, PORTD, 590, 1) ' Initialize write at sector address ' ' of 1 sector (512 bytes)</pre>
----------------	---

5.2.5.7 CF_Read_Byte – Reads 1 byte from CF

Prototype	sub function CF_READ_BYTE(dim byref CtrlPort as byte , dim byref DataPort as byte) as byte
Description	<p>Function reads 1 byte from Compact Flash. Ports need to be initialized, and CF must be initialized for reading.</p> <p>Parameters: CtrlPort - control port, DataPort - data port</p>
Example	<pre>PORTC = CF_Read_Byte(PORTB, PORTD) ' read byte and display on PORTC</pre>

5.2.5.8 CF_Read_Word – Reads 1 word from CF

Prototype	sub function CF_READ_WORD(dim byref CtrlPort as byte , dim byref DataPort as byte) as word
Description	<p>Function reads 1 word from Compact Flash. Ports need to be initialized, and CF must be initialized for reading.</p> <p>Parameters: CtrlPort - control port, DataPort - data port</p>
Example	PORTC = CF_Read_Word(PORTB, PORTD) <i>' read word and display on PORTC</i>

5.2.5.9 CF_File_Write_Init – *Initializes CF card for file writing operation (FAT16 only, PIC18 only)*

Prototype	sub procedure CF_File_Write_Init(dim byref CtrlPort as byte , dim byref DataPort as byte)
Description	<p>This procedure initializes CF card for file writing operation (FAT16 only, PIC18 only).</p> <p>Parameters: CtrlPort - control port, DataPort - data port</p>

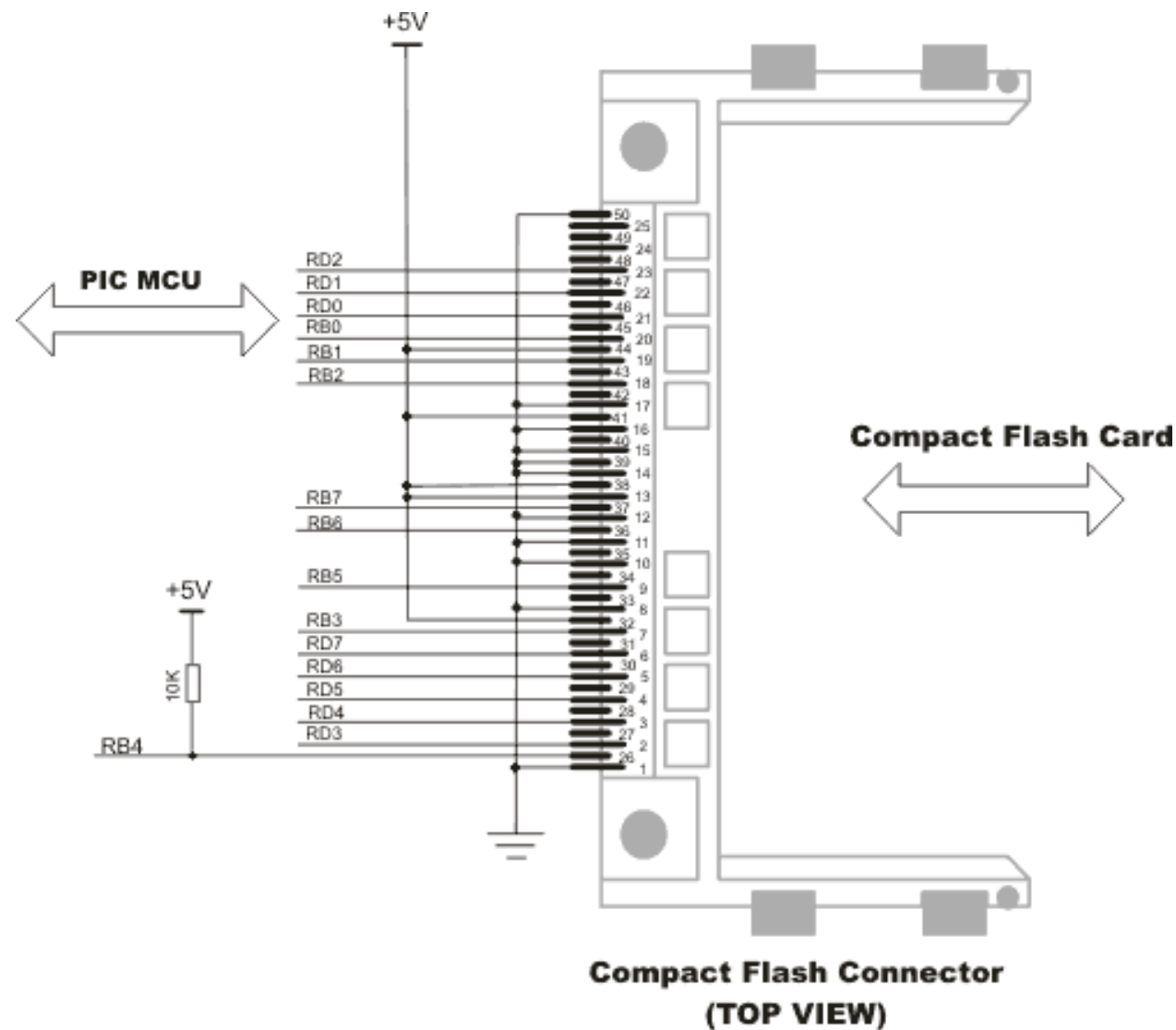
Example	<code>CF_File_Write_Init(PORTB, PORTD)</code>
----------------	---

5.2.5.10 CF_File_Write_Byte – Adds one byte to file (FAT16 only, PIC18 only)

Prototype	<code>sub procedure CF_File_Write_Byte(dim byref CtrlPort as byte, dim byref DataPort as byte, dim Bdata as byte)</code>
Description	<p>This procedure adds one byte (Bdata) to file (FAT16 only, PIC18 only).</p> <p>Parameters: CtrlPort - control port, DataPort - data port, Bdata - data byte to be written.</p>
Example	<pre>while i < 50000 CF_File_Write_Byte(PORTB, PORTD, 48 + index) ' demonstration: writes 50000 bytes to file inc(i) wend</pre>

5.2.5.11 CF_File_Write_Complete – Closes file and makes it readable (FAT16 only, PIC18 only)

Prototype	sub procedure CF_File_Write_Complete(dim byref CtrlPort as byte , dim byref DataPort as byte , dim byref Filename as char [9])
Description	<p>Upon all data has be written to file, use this procedure to close the file and make it readable by Windows (FAT16 only, PIC18 only).</p> <p>Parameters: CtrlPort - control port, DataPort - data port, Filename (must be in uppercase and must have exactly 8 characters).</p>
Example	CF_File_Write_Complete(PORTB, PORTD, "example1", "txt")



Pin diagram of CF memory card

5.2.6 EEPROM Library

EEPROM data memory is available with a number of PIC MCU models. Set of library procedures and functions is listed below to

provide you comfortable work with EEPROM.

Notes:

Be aware that all interrupts will be disabled during execution of EEPROM_Write routine (GIE bit of INTCON register will be cleared). Routine will set this bit on exit.

Ensure minimum 20ms delay between successive use of routines EEPROM_Write and EEPROM_Read. Although EEPROM will write the correct value, EEPROM_Read might return undefined result.

5.2.6.1 EEPROM_Read – *Reads 1 byte from EEPROM*

Prototype	sub function EEprom_Read(dim Address as byte) as byte
Description	<p>Function reads byte from <Address>. <Address> is of byte type, which means it can address only 256 locations. For PIC18 MCU models with more EEPROM data locations, it is programmer's responsibility to set SFR EEADRH register appropriately.</p> <p>Ensure minimum 20ms delay between successive use of routines EEPROM_Write and EEPROM_Read. Although EEPROM will write the correct value, EEPROM_Read might return undefined result.</p>

Example	<pre> TRISB = 0 Delay_ms(30) for i = 0 to 20 PORTB = EEPROM_Read(i) for j = 0 to 200 Delay_us(500) next j next i </pre>
----------------	--

5.2.6.2 EEPROM_Write – Writes 1 byte to EEPROM

Prototype	sub procedure EEprom_Write(dim Address as byte , dim Data as byte)
Description	<p>Function writes byte to <Address>. <Address> is of byte type, which means it can address only 256 locations. For PIC18 MCU models with more EEPROM data locations, it is programmer's responsibility to set SFR EEADRH register appropriately.</p> <p>All interrupts will be disabled during execution of EEPROM_Write routine (GIE bit of INTCON register will be cleared). Routine will set this bit on exit</p> <p>Ensure minimum 20ms delay between successive use of routines EEPROM_Write and EEPROM_Read. Although EEPROM will write the correct value, EEPROM_Read might return undefined result.</p>

Example	<pre> for i = 0 to 20 EEPROM_Write(i, i + 6) next i </pre>
----------------	---

5.2.7 Flash Memory Library

This library provides routines for accessing microcontroller Flash memory.

Note: Routines differ for PIC16 and PIC18 families.

5.2.7.1 Flash_Read – *Reads data from microcontroller Flash memory*

Prototype	<pre> sub function Flash_Read(dim Address as longint) as byte ' <i>for PIC18</i> sub function Flash_Read(dim Address as word) as word ' <i>for PIC16</i> </pre>
Description	Procedure reads data from the specified <Address>.
Example	<pre> for i = 0 to 63 toRead = Flash_Read(\$0D00 + i) ' <i>read 64 consecutive locations starting from 0x0D00</i> next i </pre>

5.2.7.2 Flash_Write – Writes data to microcontroller Flash memory

Prototype	<pre> sub procedure Flash_Write(dim Address as longint, dim byref Data as byte [64]) ' <i>for PIC18</i> sub procedure Flash_Write(dim Address as word, dim Data as word) ' <i>for</i> PIC16 </pre>
Description	<p>Procedure writes chunk of data to Flash memory (for PIC18, data needs to exactly 64 bytes in size). Keep in mind that this function erases target memory before writing <i><Data></i> to it. This means that if write was unsuccessful, your previous data will be lost.</p>
Example	<pre> for i = 0 to 63 ' <i>initialize array</i> toWrite[i] = i next i Flash_Write(\$0D00, toWrite) ' <i>write contents of the array to the address</i> 0x0D00 </pre>

5.2.8 I2C Library

I2C interface is serial interface used for communicating with peripheral or other microcontroller devices. Routines below are intended for PIC MCUs with MSSP module. By using these, you can configure and use PIC MCU as master in I2C communication.

5.2.8.1 I2C_Init – *Initializes I2C module*

Prototype	sub procedure I2C_Init(const Clock as longint)
Description	Initializes I2C module. Parameter <i><Clock></i> is a desired I2C clock (refer to device data sheet for correct values in respect with Fosc).
Example	I2C_Init(100000)

5.2.8.2 I2C_Start – *Issues start condition*

Prototype	sub function I2C_Start as byte
Description	Determines if I2C bus is free and issues START condition; if there is no error, function returns 0.
Example	I2C_Start

5.2.8.3 I2C_Repeated_Start – *Performs repeated start*

Prototype	sub procedure I2C_Repeated_Start
------------------	---

Description	Performs repeated start condition.
Example	I2C_Repeated_Start

5.2.8.4 I2C_Rd – *Receives byte from slave*

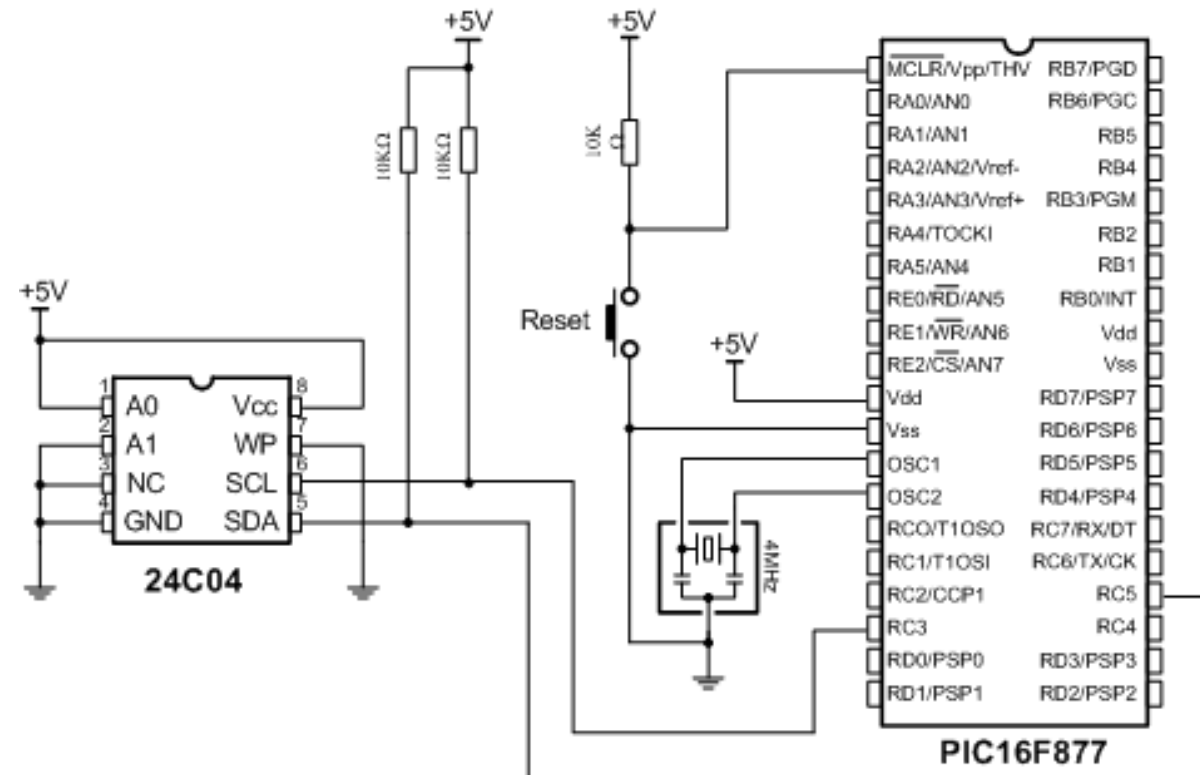
Prototype	sub function I2C_Rd(dim Ack as byte) as byte
Description	Receives 1 byte from slave and sends <i>not acknowledge</i> signal if <Ack> is 0; otherwise, it sends <i>acknowledge</i> .
Example	Data = I2C_Rd(1) ' <i>read data w/ acknowledge</i>

5.2.8.5 I2C_Wr – *Sends data byte via I2C bus*

Prototype	sub function I2C_Wr(dim Data as byte) as byte
Description	After you have issued a start or repeated start you can send <Data> byte via I2C bus. The function returns 0 if there are no errors.
Example	I2C_Wr(\$A2) ' <i>send byte via I2C(command to 24c02)</i>

5.2.8.6 I2C_Stop – Issues STOP condition

Prototype	<code>sub procedure I2C_Stop as byte</code>
Description	Issues STOP condition.
Example	I2C_Stop



Example of I2C communication with 24c02 EEPROM

5.2.9 LCD Library

BASIC provides a set of library procedures and functions for communicating with commonly used 4-bit interface LCD (with Hitachi HD44780 controller). Be sure to designate port with LCD as output, before using any of the following library procedures or functions.

5.2.9.1 LCD_Init – *Initializes LCD with default pin settings*

Prototype	<code>sub procedure LCD_Init(dim byref Port as byte)</code>
Description	Initializes LCD at <Port> with default pin settings (see the figure below).
Example	<pre>LCD_Init(PORTB) ' Initializes LCD on PORTB (check pin settings in the figure below)</pre>

5.2.9.2 LCD_Config – *Initializes LCD with custom pin settings*

Prototype	<code>sub procedure LCD_Config(dim byref Port as byte, const RS, const EN, const WR, const D7, const D6, const D5, const D4)</code>
------------------	---

Description	Initializes LCD at <i><Port></i> with pin settings you specify: parameters <i><RS></i> , <i><EN></i> , <i><WR></i> , <i><D7></i> .. <i><D4></i> need to be a combination of values 0..7 (e.g. 3,6,0,7,2,1,4).
Example	<pre>LCD_Config(PORTD, 1, 2, 0, 3, 5, 4, 6)</pre> <p><i>' Initializes LCD on PORTD with our custom pin settings</i></p>

5.2.9.3 LCD_Chr – Prints char on LCD at specified row and col

Prototype	sub procedure LCD_Chr(dim Row as byte , dim Column as byte , dim Character as byte)
Description	Prints <i><Character></i> at specified <i><Row></i> and <i><Column></i> on LCD.
Example	<pre>LCD_Chr(1, 2, "e")</pre> <p><i>' Prints character "e" on LCD (1st row, 2nd column)</i></p>

5.2.9.4 LCD_Chr_CP – Prints char on LCD at current cursor position

Prototype	sub procedure LCD_Chr_CP(dim Character as byte)
Description	Prints <i><Character></i> at current cursor position.

Example	<pre>LCD_Chr_CP("k")</pre> <p><i>' Prints character "k" at current cursor position</i></p>
----------------	--

5.2.9.5 LCD_Out – Prints string on LCD at specified row and col

Prototype	sub procedure LCD_Out(dim Row as byte, dim Column as byte, dim byref Text as char[255])
Description	Prints <i><Text></i> (string variable) at specified <i><Row></i> and <i><Column></i> on LCD. Both string variables and string constants can be passed.
Example	<pre>LCD_Out(1, 3, Text)</pre> <p><i>' Prints string variable Text on LCD (1st row, 3rd column)</i></p>

5.2.9.6 LCD_Out_CP – Prints string on LCD at current cursor position

Prototype	sub procedure LCD_Out_CP(dim byref Text as char[255])
Description	Prints <i><Text></i> (string variable) at current cursor position. Both string variables and string constants can be passed.

Example	<pre>LCD_Out_CP("Some text") ' Prints "Some text" at current cursor position</pre>
----------------	--

5.2.9.7 LCD_Cmd – Sends command to LCD

Prototype	sub procedure LCD_Cmd(dim Command as byte)
Description	<p>Sends <Command> to LCD.</p> <p>List of available commands follows:</p> <pre>LCD_First_Row ' Moves cursor to 1st row LCD_Second_Row ' Moves cursor to 2nd row LCD_Third_Row ' Moves cursor to 3rd row LCD_Fourth_Row ' Moves cursor to 4th row LCD_Clear</pre>

```
' Clears display
```

```
LCD_Return_Home
```

```
' Returns cursor to home position,  
' returns a shifted display to original position.  
' Display data RAM is unaffected.
```

```
LCD_Cursor_Off
```

```
' Turn off cursor
```

```
LCD_Underline_On
```

```
' Underline cursor on
```

```
LCD_Blink_Cursor_On
```

```
' Blink cursor on
```

```
LCD_Move_Cursor_Left
```

```
' Move cursor left without changing display data RAM
```

```
LCD_Move_Cursor_Right
```

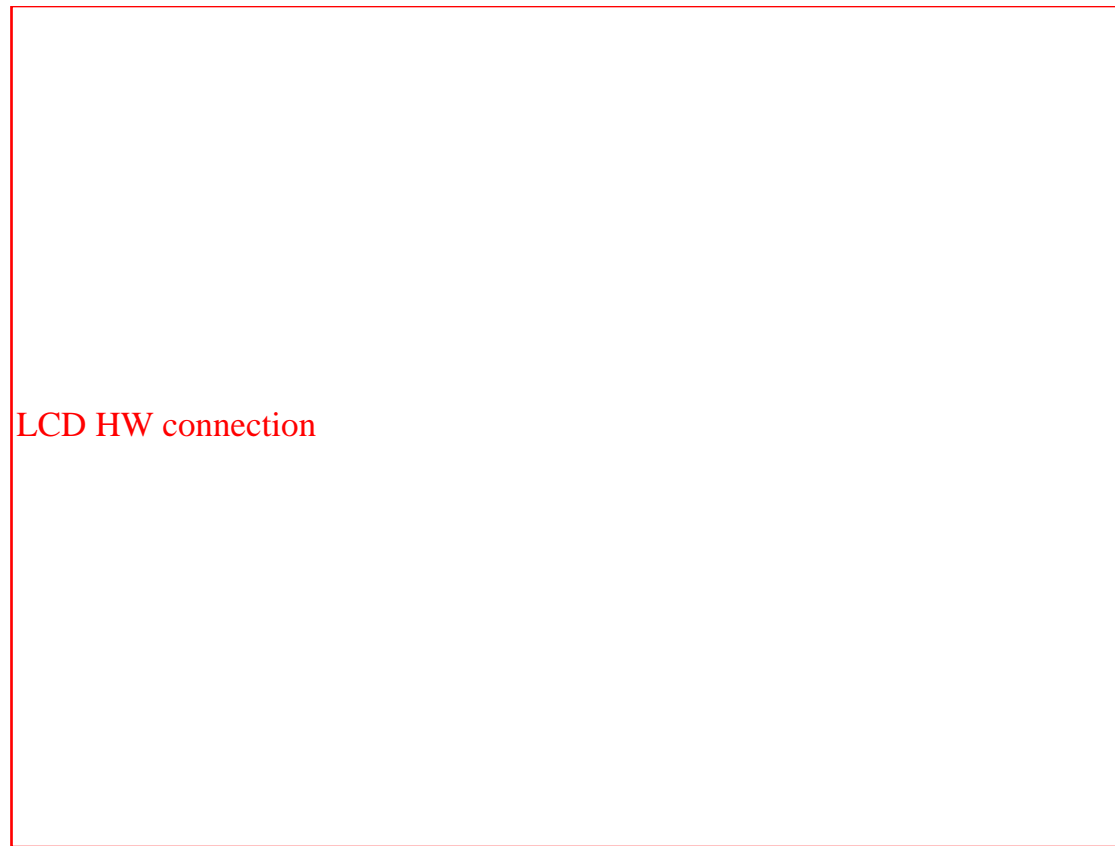
```
' Move cursor right without changing display data RAM
```

```
LCD_Turn_On
```

```
' Turn LCD display on
```

```
LCD_Turn_Off
```

	<pre> ' Turn LCD display off LCD_Shift_Left ' Shift display left without changing display data RAM LCD_Shift_Right ' Shift display right without changing display data RAM </pre>
Example	<pre> LCD_Cmd(LCD_Clear) ' Clears LCD display </pre>



LCD HW connection

LCD HW connection

5.2.10 LCD8 Library (8-bit interface LCD)

BASIC provides a set of library procedures and functions for communicating with commonly used 8-bit interface LCD (with Hitachi HD44780 controller). Be sure to designate Control and Data ports with LCD as output, before using any of the following library procedures or functions.

5.2.10.1 LCD8_Init – *Initializes LCD with default pin settings*

Prototype	sub procedure LCD8_Init(dim byref Port_Ctrl as byte , dim byref Port_Data as byte)
Description	Initializes LCD at <Port_Ctrl> and <Port_Data> with default pin settings (see the figure below).
Example	<pre>LCD8_Init(PORTB, PORTC) ' Initializes LCD on PORTB and PORTC with default pin settings ' (check pin settings in the figure below)</pre>

5.2.10.2 LCD8_Config – Initializes LCD with custom pin settings

Prototype	sub procedure LCD8_Config(dim byref Port_Ctrl as byte , dim byref Port_Data as byte , const RS, const EN, const WR, const D7, const D6, const D5, const D4, const D3, const D2, const D1, const D0)
Description	Initializes LCD at <Port_Ctrl> and <Port_Data> with pin settings you specify: parameters <RS>, <EN>, <WR> need to be in range 0..7; parameters <D7>..<D0> need to be a combination of values 0..7 (e.g. 3,6,5,0,7,2,1,4).
Example	<pre>LCD8_Config(PORTC, PORTD, 0, 1, 2, 6, 5, 4, 3, 7, 1, 2, 0) ' Initializes LCD on PORTC and PORTD with our custom pin settings</pre>

5.2.10.3 LCD8_Chrc – Prints char on LCD at specified row and col

Prototype	sub procedure LCD8_Chrc(dim Row as byte , dim Column as byte , dim Character as byte)
Description	Prints <i><Character></i> at specified <i><Row></i> and <i><Column></i> on LCD.
Example	<pre>LCD8_Chrc(1, 2, "e") ' Prints character "e" on LCD (1st row, 2nd column)</pre>

5.2.10.4 LCD8_Chrc_CP – Prints char on LCD at current cursor position

Prototype	sub procedure LCD8_Chrc_CP(dim Character as byte)
Description	Prints <i><Character></i> at current cursor position.
Example	<pre>LCD8_Chrc_CP("k") ' Prints character "k" at current cursor position</pre>

5.2.10.5 LCD8_Out – Prints string on LCD at specified row and col

Prototype	sub procedure LCD8_Out(dim Row as byte, dim Column as byte, dim byref Text as char[255])
Description	Prints <Text> (string variable) at specified <Row> and <Column> on LCD. Both string variables and string constants can be passed.
Example	<pre>LCD8_Out(1, 3, Text) ' Prints string variable Text on LCD (1st row, 3rd column)</pre>

5.2.10.6 LCD8_Out_CP – Prints string on LCD at current cursor position

Prototype	sub procedure LCD8_Out_CP(dim byref Text as char[255])
Description	Prints <Text> (string variable) at current cursor position. Both string variables and string constants can be passed.
Example	<pre>LCD8_Out_CP("Test") ' Prints "Test" at current cursor position</pre>

5.2.10.7 LCD8_Cmd – Sends command to LCD

Prototype	sub procedure LCD8_Cmd(dim Command as byte)
Description	<p>Sends <i><Command></i> to LCD.</p> <p>List of available commands follows:</p> <p>LCD_First_Row ' Moves cursor to 1st row</p> <p>LCD_Second_Row ' Moves cursor to 2nd row</p> <p>LCD_Third_Row ' Moves cursor to 3rd row</p> <p>LCD_Fourth_Row ' Moves cursor to 4th row</p> <p>LCD_Clear ' Clears display</p> <p>LCD_Return_Home ' Returns cursor to home position, ' returns a shifted display to original position. ' Display data RAM is unaffected.</p>

LCD_Cursor_Off

' Turn off cursor

LCD_Underline_On

' Underline cursor on

LCD_Blink_Cursor_On

' Blink cursor on

LCD_Move_Cursor_Left

' Move cursor left without changing display data RAM

LCD_Move_Cursor_Right

' Move cursor right without changing display data RAM

LCD_Turn_On

' Turn LCD display on

LCD_Turn_Off

' Turn LCD display off

LCD_Shift_Left

' Shift display left without changing display data RAM

LCD_Shift_Right

' Shift display right without changing display data RAM

Example	<code>LCD8_Cmd(LCD_Clear)</code> <i>' Clears LCD display</i>
----------------	--



LCD HW connection

LCD HW connection

5.2.11 Graphic LCD Library

mikroPascal provides a set of library procedures and functions for drawing and writing on Graphical LCD. Also it is possible to

convert bitmap (use menu option Tools > BMP2LCD) to constant array and display it on GLCD. These routines works with commonly used GLCD 128x64, and work only with the PIC18 family.

5.2.11.1 GLCD_Config – *Initializes GLCD with custom pin settings*

Prototype	<code>sub procedure GLCD_Config(dim byref Ctrl_Port as byte, dim byref Data_Port as byte, dim Reset as byte, dim Enable as byte,dim RS as byte, dim RW as byte, dim CS1 as byte, dim CS2 as byte)</code>
Description	Initializes GLCD at <Ctrl_Port> and <Data_Port> with custom pin settings.
Example	<code>GLCD_LCD_Config(PORTB, PORTC, 1,7,4,6,0,2)</code>

5.2.11.2 GLCD_Init – *Initializes GLCD with default pin settings*

Prototype	<code>sub procedure GLCD_Init(dim Ctrl_Port as byte, dim Data_Port as byte)</code>
Description	Initializes LCD at <Ctrl_Port> and <Data_Port>. With default pin settings Reset=7, Enable=1, RS=3, RW=5, CS1=2, CS2=0.
Example	<code>GLCD_LCD_Init(PORTB, PORTC)</code>

5.2.11.3 GLCD_Put_Ins – Sends instruction to GLCD.

Prototype	sub procedure GLCD_Put_Ins(dim Ins as byte)
Description	<p>Sends instruction <Ins> to GLCD. Available instructions include:</p> <pre> X_ADDRESS = \$B8 ' Adress base for Page 0 Y_ADDRESS = \$40 ' Adress base for Y0 START_LINE = \$C0 ' Adress base for line 0 DISPLAY_ON = \$3F ' Turn display on DISPLAY_OFF = \$3E ' Turn display off </pre>
Example	GLCD_Put_Ins(DISPLAY_ON)

5.2.11.4 GLCD_Put_Data – Sends data byte to GLCD.

Prototype	sub procedure GLCD_Put_Data(dim data as byte)
Description	Sends data byte to GLCD.

Example	<code>GLCD_Put_Data(temperature)</code>
----------------	---

5.2.11.5 GLCD_Put_Data2 – Sends data byte to GLCD.

Prototype	<code>sub procedure GLCD_Put_Data2(dim data as byte, dim side as byte)</code>
Description	Sends data to GLCD at specified <i><side></i> (<i><side></i> can take constant value LEFT or RIGHT) .
Example	<code>GLCD_Put_Data2(temperature, 1)</code>

5.2.11.6 GLCD_Select_Side- Selects the side of the GLCD.

Prototype	<code>sub procedure GLCD_Select_Side(dim LCDSide as byte)</code>
Description	Selects the side of the GLCD: <pre>' const RIGHT = 0 ' const LEFT = 1</pre>
Example	<code>GLCD_Select_Side(1)</code>

5.2.11.7 GLCD_Data_Read – *Reads data from GLCD.*

Prototype	sub function GLCD_Data_Read as byte
Description	Reads data from GLCD.
Example	GLCD_Data_Read

5.2.11.8 GLCD_Clear_Dot – *Clears a dot on the GLCD.*

Prototype	sub procedure GLCD_Clear_Dot(dim x as byte, dim y as byte)
Description	Clears a dot on the GLCD at specified coordinates.
Example	GLCD_Clear_Dot(20, 32)

5.2.11.9 GLCD_Set_Dot – *Draws a dot on the GLCD.*

Prototype	sub procedure GLCD_Set_Dot(dim x as byte, dim y as byte)

Description	Draws a dot on the GLCD at specified coordinates.
Example	GLCD_Set_Dot(20 , 32)

5.2.11.10 GLCD_Circle – *Draws a circle on the GLCD.*

Prototype	sub procedure GLCD_Circle(dim CenterX as integer , dim CenterY as integer , dim Radius as integer)
Description	Draws a circle on the GLCD, centered at <CenterX, CenterY> with <Radius>.
Example	GLCD_Circle(30, 42, 6)

5.2.11.11 GLCD_Line – *Draws a line*

Prototype	sub procedure GLCD_Line(dim x1 as integer , dim y1 as integer , dim x2 as integer , dim y2 as integer)
Description	Draws a line from (x1,y1) to (x2,y2).

Example	<pre>GLCD_Line(0, 0, 120, 50) GLCD_Line(0,63, 50, 0)</pre>
----------------	--

5.2.11.12 GLCD_Invert – *Inverts display*

Prototype	sub procedure GLCD_Invert(dim Xaxis as byte , dim Yaxis as byte)
Description	Procedure inverts display (changes dot state on/off) in the specified area, X pixels wide starting from 0 position, 8 pixels high. Parameter Xaxis spans 0..127, parameter Yaxis spans 0..7 (8 text lines).
Example	GLCD_Invert(60, 6)

5.2.11.13 GLCD_Goto_XY – *Sets cursor to dot(x,y)*

Prototype	sub procedure GLCD_Goto_XY(dim x as byte , dim y as byte)
Description	Sets cursor to dot (x,y). Procedure is used in combination with GLCD_Put_Data, GLCD_Put_Data2, and GLCD_Put_Char.
Example	GLCD_Goto_XY(60, 6)

5.2.11.14 GLCD_Put_Char – *Prints <Character> at cursor position*

Prototype	sub procedure GLCD_Put_Char(dim Character as byte)
Description	Prints <Character> at cursor position.
Example	GLCD_Put_Char(k)

5.2.11.15 GLCD_Clear_Screen – *Clears the GLCD screen*

Prototype	sub procedure GLCD_Clear_Screen
Description	Clears the GLCD screen.
Example	GLCD_Clear_Screen

5.2.11.16 GLCD_Put_Text – *Prints text at specified position*

Prototype	sub procedure GLCD_Put_Text(dim x_pos as word , dim y_pos as word , dim byref text as char [25], dim invert as byte)
Description	Prints <text> at specified position; y_pos spans 0..7.
Example	GLCD_Put_Text(0, 7, My_text, NONINVERTED_TEXT)

5.2.11.17 GLCD_Rectangle – *Draws a rectangle*

Prototype	sub procedure GLCD_Rectangle(dim X1 as byte , dim Y1 as byte , dim X2 as byte , dim Y2 as byte)
Description	Draws a rectangle on the GLCD. (x1,y1) sets the upper left corner, (x2,y2) sets the lower right corner.
Example	GLCD_Rectangle(10, 0, 30, 35)

5.2.11.18 GLCD_Set_Font – *Sets font for GLCD*

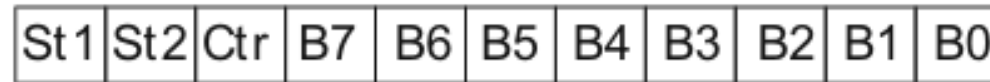
Prototype	sub procedure GLCD_Set_Font(dim font_index as byte)

Description	Sets font for GLCD. Parameter <i><font_index></i> spans from 1 to 4, and determines which font will be used: 1: 5x8 dots 2: 5x7 3: 3x6 4: 8x8
Example	GLCD_Set_Font (2)

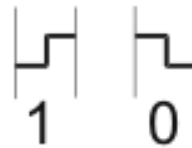
5.2.12 Manchester Code Library

mikroBasic provides a set of library procedures and functions for handling Manchester coded signal. Manchester code is a code in which data and clock signals are combined to form a single self-synchronizing data stream; each encoded bit contains a transition at the midpoint of a bit period, the direction of transition determines whether the bit is a 0 or a 1; second half is the true bit value and the first half is the complement of the true bit value (as shown in the figure below).

Manchester RF_Send_Byte format

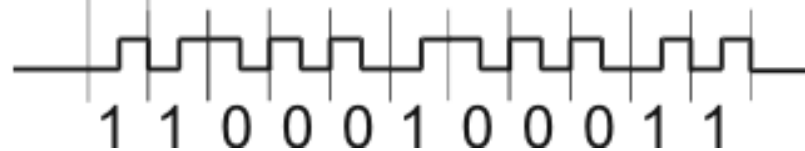


Bi-phase coding



2.4ms

Example of transmission



Note: Manchester receive routines are blocking calls (Man_Receive_Config, Man_Receive_Init, Man_Receive). This means that PIC will wait until the task is performed (e.g. byte is received, synchronization achieved, etc).

Note: Routines for receiving are limited to a baud rate scope from 340 ~ 560 bps.

5.2.12.1 Man_Receive_Init – *Initialization with default pin*

Prototype	<code>sub procedure Man_Receive_Init(dim byref Port as byte)</code>
Description	Procedure works same as Man_Receive_Config, but with default pin setting (pin 6).
Example	<code>Man_Receive_Init(PORTD)</code>

5.2.12.2 Man_Receive_Config – Initialization with custom pin

Prototype	sub procedure Man_Receive_Config(dim byref Port as byte , dim RXpin as byte)
Description	This procedure needs to be called in order to receive signal by procedure Man_Receive. You need to specify the <i><Port></i> and <i><RXpin></i> of input signal. In case of multiple errors on reception, you should call Man_Receive_Init once again to enable synchronization.
Example	Man_Receive_Config(PORTD, 5)

5.2.12.3 Man_Receive – Receives a byte

Prototype	sub function Man_Receive(dim byref Error as byte) as byte
Description	Function extracts one byte from signal. If format does not match the expected, <i><Error></i> flag will be set True.
Example	<pre> dim ErrorFlag as byte temp = Man_Receive(ErrorFlag) ' Attempt byte receive </pre>

5.2.12.4 Man_Send_Init – Initialization with default pin

Prototype	sub procedure Man_Send_Init(dim byref Port as byte)
Description	Procedure works same as Man_Send_Config, but with default pin setting (pin 0).
Example	Man_Send_Init(PORTB)

5.2.12.5 Man_Send_Config – Initialization with custom pin

Prototype	sub procedure Man_Send_Config(dim byref Port as byte , dim TXpin as byte)
Description	Procedure needs to be called in order to send signals via procedure Man_Send. Procedure specifies <i><Port></i> and <i><TXpin></i> for outgoing signal (const baud rate).
Example	Man_Send_Config(PORTB, 4)

5.2.12.6 Man_Send – Sends a byte

Prototype	sub procedure Man_Send(dim Data as byte)
Description	Procedure sends one <i><Data></i> byte.

Example	<pre> for i = 1 to Length(s1) Man_Send(s1[i]) ' <i>Send char</i> Delay_ms(90) next i </pre>
----------------	--

5.2.13 PWM Library

CCP (Capture/ Compare/ PWM) module is available with a number of PIC MCU models. Set of library procedures and functions is listed below to provide comfortable work with PWM (Pulse Width Modulation).

Note that these routines support module on PORTC pin RC2, and won't work with modules on other ports. Also, BASIC doesn't support enhanced PWM modules.

5.2.13.1 PWM_Init – *Initializes PWM module*

Prototype	sub procedure PWM_Init(const PWM_Freq)
Description	Initializes PWM module with (duty ratio) 0%. <i><PWM_Freq></i> is a desired PWM frequency (refer to device data sheet for correct values in respect with Fosc).
Example	<pre> PWM_Init(5000) ' <i>initializes PWM module, freq = 5kHz</i> </pre>

5.2.13.2 PWM_Change_Duty – *Changes duty ratio*

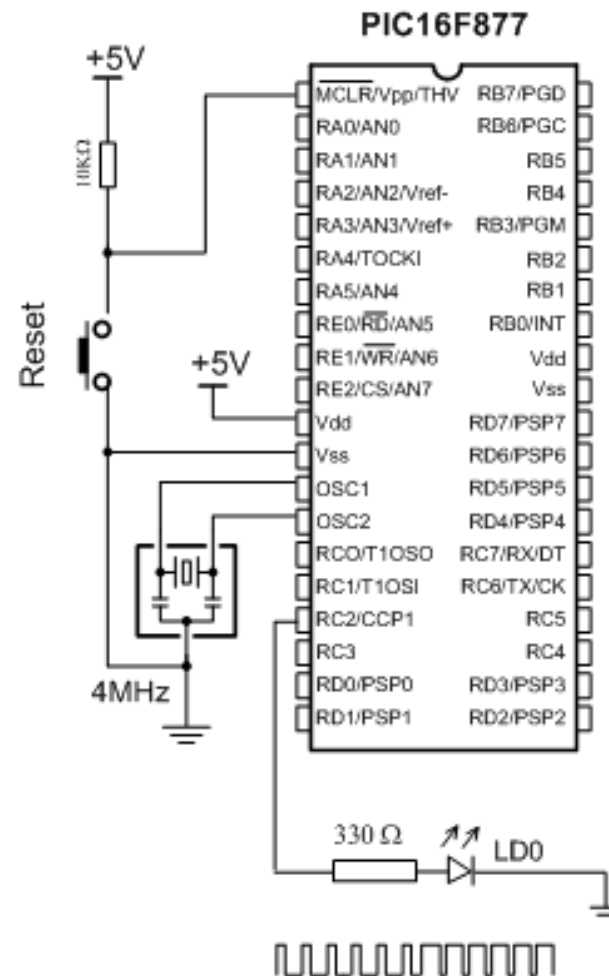
Prototype	sub procedure PWM_Change_Duty(dim New_Duty as byte)
Description	Routine changes duty ratio. <New_Duty> takes values from 0 to 255, where 0 is 0% duty ratio, 127 is 50% duty ratio, and 255 is 100% duty ratio. Other values for specific duty ratio can be calculated as (Percent*255)/100.
Example	<pre> while true Delay_ms(100) j = j + 1 PWM_Change_Duty(j) wend </pre>

5.2.13.3 PWM_Start – *Starts PWM*

Prototype	sub procedure PWM_Start
Description	Starts PWM.
Example	PWM_Start

5.2.13.4 PWM_Stop – *Stops PWM*

Prototype	<code>sub procedure PWM_Stop</code>
Description	Stops PWM.
Example	<code>PWM_Stop</code>



PWM demonstration

5.2.14 RS485 Library

RS485 is a multipoint communication which allows multiple devices to be connected to a single signal cable. BASIC provides a set of library routines to provide you comfortable work with RS485 system using Master/Slave architecture.

Master and Slave devices interchange packets of information, each of these packets containing synchronization bytes, CRC byte, address byte, and the data. In Master/Slave architecture, Slave can never initiate communication. Each Slave has its unique address and receives only the packets containing that particular address. It is programmer's responsibility to ensure that only one device transmits data via 485 bus at a time.

RS485 routines require USART module on port C. Pins of USART need to be attached to RS485 interface transceiver, such as LTC485 or similar. Pins of transceiver (Receiver Output Enable and Driver Outputs Enable) should be connected to port C, pin 2 (see the figure at end of the chapter).

Note: Address 50 is a common address for all Slave devices: packets containing address 50 will be received by all Slaves. The only exceptions are Slaves with addresses 150 and 169, which require their particular address to be specified in the packet.

5.2.14.1 RS485Master_Init – *Initializes MCU as Master in RS485 communication*

Prototype	sub procedure RS485master_init
Description	Initializes MCU as Master in RS485 communication. USART needs to be initialized.
Example	RS485Master_Init

5.2.14.2 RS485Master_Read – *Receives message from Slave*

Prototype	sub procedure RS485master_read(dim byref data as byte [5])
Description	<p>Master receives any message sent by Slaves. As messages are multi-byte, this procedure must be called for each byte received (see the example at the end of the chapter). Upon receiving a message, buffer is filled with the following values:</p> <ul style="list-style-type: none"> • data[0..2] is actual data • data[3] is number of bytes received, 1..3 • data[4] is set to 255 when message is received • data[5] is set to 255 if error has occurred • data[6] is the address of the Slave which sent the message <p>Procedure automatically sets data[4] and data[5] upon every received message. These flags need to be cleared repeatedly from the program.</p> <p>Note: MCU must be initialized as Master in 485 communication to assign an address to MCU</p>
Example	RS485Master_Read(dat)

5.2.14.3 RS485Master_Write – Sends message to Slave

Prototype	sub procedure RS485Master_Write(dim byref data as byte [2], dim datalen as byte , dim address as byte)
------------------	---

Description	<p>Routine sends number of bytes ($1 < \text{datalen} \leq 3$) from buffer via 485, to slave specified by <i><address></i>.</p> <p>MCU must be initialized as Master in 485 communication. It is programmer's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.</p>
Example	<code>RS485Master_Write(dat, 1)</code>

5.2.14.4 RS485Slave_Init – *Initializes MCU as Slave in RS485 communication*

Prototype	sub procedure RS485Slave_Init(dim address as byte)
Description	<p>Initializes MCU as Slave in RS485 communication. USART needs to be initialized.</p> <p><i><address></i> can take any value between 0 and 255, except 50, which is common address for all slaves.</p>
Example	<code>RS485Slave_Init(160)</code> ' initialize MCU as Slave with address 160

5.2.14.5 RS485Slave_Read – *Receives message from Master*

Prototype	sub procedure RS485Slave_Read(dim byref data as byte[5])
------------------	--

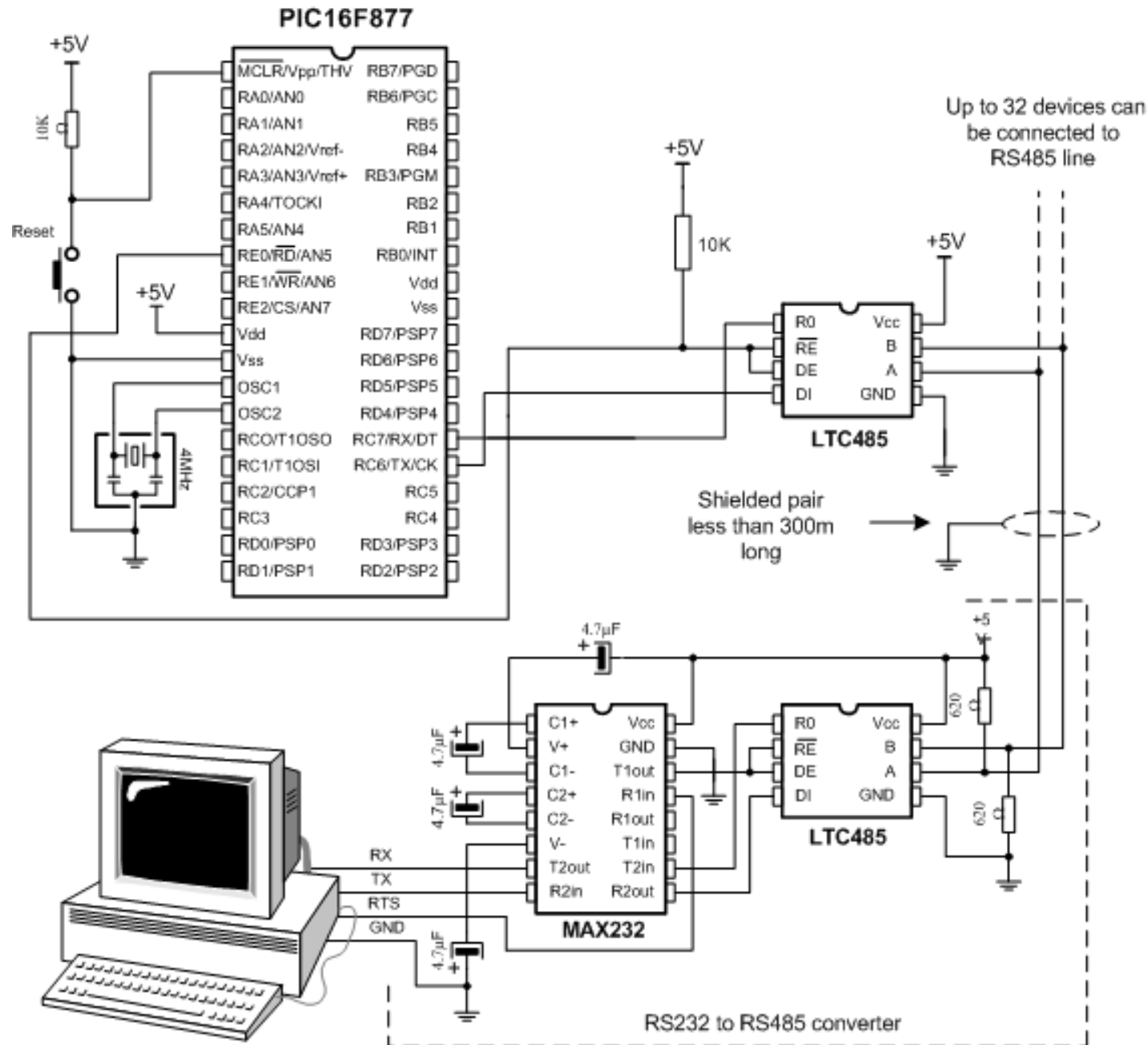
Description	<p>Only messages that appropriately address Slaves will be received. As messages are multi-byte, this procedure must be called for each byte received (see the example at the end of the chapter). Upon receiving a message, buffer is filled with the following values:</p> <ul style="list-style-type: none"> • data[0..2] is actual data • data[3] is number of bytes received, 1..3 • data[4] is set to 255 when message is received • data[5] is set to 255 if error has occurred • rest of the buffer is undefined <p>Procedure automatically sets data[4] and data[5] upon every received message. These flags need to be cleared repeatedly from the program.</p> <p>MCU must be initialized as Master in 485 communication to assign an address to MCU.</p>
Example	RS485Slave_Read(dat)

5.2.14.6 RS485Slave_Write – Sends message to Master

Prototype	<pre>sub procedure RS485Slave_Write(dim byref data as byte[2], dim datalen as byte)</pre>

Description	<p>Sends number of bytes ($1 < \text{datalen} \leq 3$) from buffer via 485 to Master.</p> <p>MCU must be initialized as Slave in 485 communication. It is programmer's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.</p>
Example	<pre>RS485Slave_Write(dat, 1)</pre>

Connecting PC and PIC via RS485 communication line



Example of interfacing PC to PIC MCU via RS485 bus

5.2.15 SPI Library

SPI (Serial Peripheral Interface) module is available with a number of PIC MCU models. You can easily communicate with other devices via SPI - A/D converters, D/A converters, MAX7219, LTC1290 etc. You need PIC MCU with hardware integrated SPI (for example, PIC16F877). Then, simply use the following functions and procedures.

5.2.15.1 SPI_Init – *Standard initialization of SPI*

Prototype	<code>sub procedure SPI_Init</code>
Description	<p>Routine initializes SPI with default parameters:</p> <ul style="list-style-type: none"> • Master mode, • clock Fosc/4, • clock idle state low, • data transmitted on low to high edge, • input data sampled at the middle of interval.
Example	<code>SPI_Init</code>

5.2.15.2 SPI_Init_Advanced – *does smt*

Prototype	sub procedure SPI_Init_Advanced(dim Master as byte , dim Data_Sample as byte , dim Clock_Idle as byte , dim Low_To_High as byte)
Description	<p>For advanced settings, configure and initialize SPI using the procedure SPI_Init_Advanced.</p> <p>Allowed values of parameters:</p> <p><Master> determines the work mode for SPI:</p> <ul style="list-style-type: none"> • Master_OSC_div4 : Master clock=Fosc/4 • Master_OSC_div16 : Master clock=Fosc/16 • Master_OSC_div64 : Master clock=Fosc/64 • Master_TMR2 : Master clock source TMR2 • Slave_SS_ENABLE : Master Slave select enabled • Slave_SS_DIS : Master Slave select disabled <p><Data_Sample> determines when data is sampled:</p> <ul style="list-style-type: none"> • Data_SAMPLE_MIDDLE : input data sampled in middle of interval • Data_SAMPLE_END : input data sampled at the end of interval <p><Clock_Idle> determines idle state for clock:</p> <ul style="list-style-type: none"> • CLK_Idle_HIGH : clock idle HIGH • CLK_Idle_LOW : clock idle LOW

<Low_To_High> determines transmit edge for data:

- LOW_2_HIGH : data transmit on low to high edge
- HIGH_2_LOW : data transmit on high to low edge

Example

```
SPI_Init_Advanced(Master_OSC_div4, Data_SAMPLE_MIDDLE, CLK_Idle_LOW,
LOW_2_HIGH)
```

```
' This will set SPI to:
' master mode,
' clock = Fosc/4,
' data sampled at the middle of interval,
' clock idle state low,
' data transmitted at low to high edge.
```

5.2.15.3 SPI_Read – Reads the received data

Prototype	sub function SPI_Read(dim Buffer as byte) as byte
Description	Routine provides clock by sending <Buffer> and reads the received data at the end of the period.

Example	<pre> dim rec as byte ... SPI_Read(rec) </pre>
----------------	--

5.2.15.4 SPI_Write – Sends data via SPI

Prototype	<code>sub procedure SPI_Write(dim Data as byte)</code>
Description	Routine writes <i><Data></i> to SSPBUF and immediately starts the transmission.
Example	<code>SPI_Write(7)</code>

5.2.16 USART Library

USART (Universal Synchronous Asynchronous Receiver Transmitter) hardware module is available with a number of PIC MCU models. You can easily communicate with other devices via RS232 protocol (for example with PC, see the figure at the end of this chapter - RS232 HW connection). You need a PIC MCU with hardware integrated USART (for example, PIC16F877). Then, simply use the functions and procedures described below.

Note: Some PIC micros that have two USART modules, such as P18F8520, require you to specify the module you want to use. Simply append the number 1 or 2 to procedure or function name, e.g. `USART_Write2(Dat)`.

5.2.16.1 USART_Init – *Initializes USART*

Prototype	sub procedure USART_Init(const Baud_Rate)
Description	<p>Initializes PIC MCU USART hardware and establishes communication at specified <i><Baud_Rate></i>.</p> <p>Refer to the device data sheet for baud rates allowed for specific Fosc. If you specify the unsupported baud rate, compiler will report an error.</p>
Example	USART_Init(2400)

5.2.16.2 USART_Data_Ready – *Checks if data is ready*

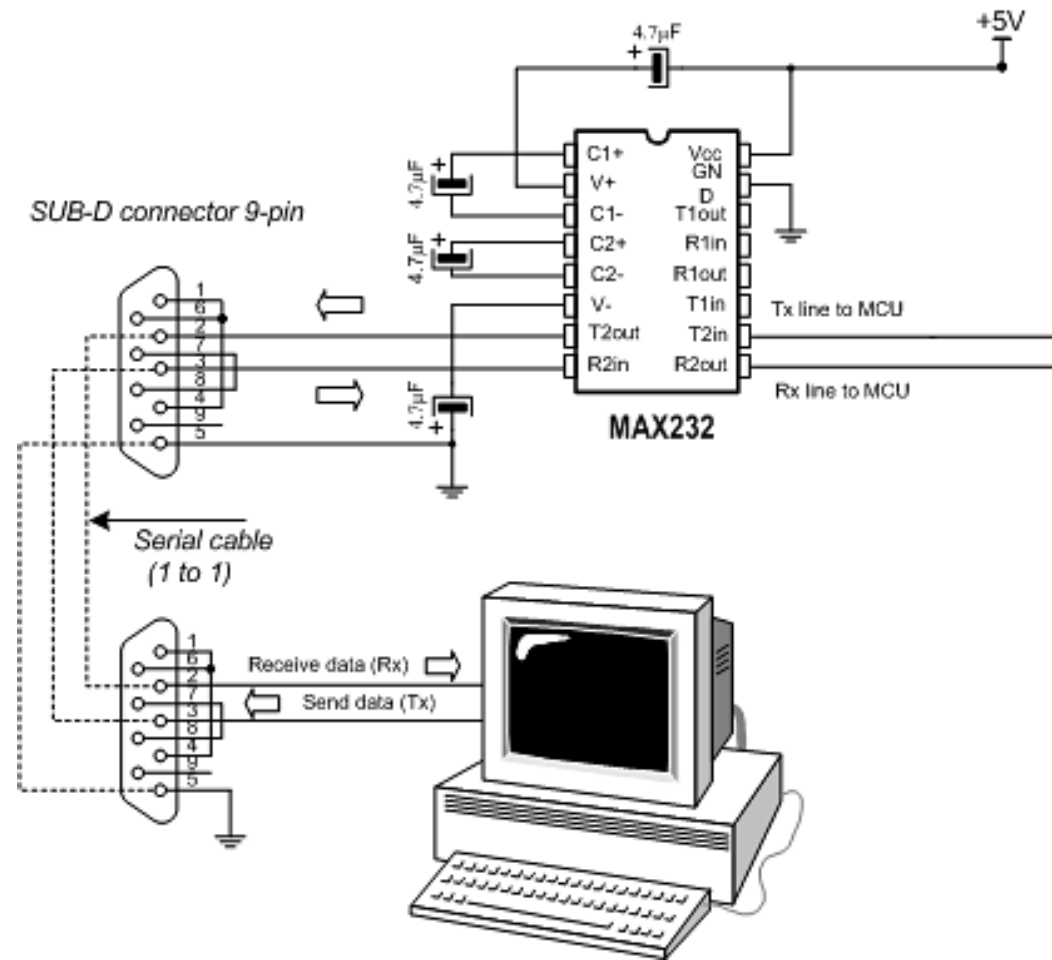
Prototype	sub function USART_Data_Ready as byte
Description	Function checks if data is ready. Returns 1 if so, returns 0 otherwise.
Example	USART_Data_Ready

5.2.16.3 USART_Read – *Receives a byte*

Prototype	sub function USART_Read as byte
Description	Receives a byte; if byte is not received returns 0.
Example	USART_Read

5.2.16.4 USART_Write – *Transmits a byte*

Prototype	sub procedure USART_Write(dim Data as byte)
Description	Procedure transmits byte <i><Data></i> .
Example	USART_Write(dat)



RS232 HW connection

5.2.17 One-Wire Library

1-wire library provides routines for communicating via 1-wire bus, for example with DS1820 digital thermometer. Note that oscillator frequency F_{osc} needs to be at least 4MHz in order to use the routines with Dallas digital thermometers.

5.2.17.1 OW_Reset – Issues 1-wire reset signal for DS1820

Prototype	<code>sub function OW_Reset(dim byref PORT as byte, dim Pin as byte) as byte</code>
Description	Issues 1-wire reset signal for DS1820. Parameters <i><PORT></i> and <i><Pin></i> specify the location of DS1820; return value of the function is 0 if DS1820 is present, and 1 otherwise.
Example	<code>OW_Reset(PORTA, 5)</code>

5.2.17.2 OW_Read – Reads one byte via 1-wire bus

Prototype	<code>sub function OW_Read(dim byref PORT as byte, Pin as byte) as byte</code>
Description	Reads one byte via 1-wire bus.
Example	<code>temp = OW_Read(PORTA, 5) ' get result from PORTA</code>

5.2.17.3 OW_Write – Writes one byte via 1-wire bus

Prototype	sub procedure OW_Write(dim byref PORT as byte , dim Pin as byte , dim par as byte)
Description	Writes one byte (<par>) via 1-wire bus
Example	OW_Write(PORTA, 5, \$44)

5.2.18 Software I2C

BASIC provides routines which implement software I2C. These routines are hardware independent and can be used with any MCU. Software I2C enables you to use MCU as Master in I2C communication. Multi-master mode is not supported.

5.2.18.1 Soft_I2C_Config – *Configure the I2C master mode*

Prototype	sub procedure Soft_I2C_Config(dim byref Port as byte , const SDA, const SCL)
Description	Configure the I2C master mode. Parameter <Port> specifies port of MCU on which SDA and SCL pins will be located; parameters <SCL> and <SDA> need to be in range 0..7 and cannot point at the same pin;
Example	Soft_I2C_Config(PORTD, 3, 4)

5.2.18.2 Soft_I2C_Start – *Issues START condition*

Prototype	sub procedure Soft_I2C_Start
Description	Issues START condition.
Example	Soft_I2C_Start

5.2.18.3 Soft_I2C_Write – *Send data byte via I2C bus*

Prototype	sub function Soft_I2C_Write(dim Data as byte) as byte
Description	After you have issued a start signal you can send <i><Data></i> byte via I2C bus. The function returns 0 if there are no errors.
Example	Soft_I2C_Write(\$A3)

5.2.18.4 Soft_I2C_Read – *Receives byte from slave*

Prototype	sub function Soft_I2C_Read(dim Ack as byte) as byte
Description	Receives 1 byte from slave and sends <i>not acknowledge</i> signal if <Ack> is 0; otherwise, it sends <i>acknowledge</i> .
Example	EE_data = Soft_I2C_Read(0)

5.2.18.5 Soft_I2C_Stop – Issues STOP condition

Prototype	sub procedure Soft_I2C_Stop
Description	Issues STOP condition.
Example	Soft_I2C_Stop

5.2.19 Software SPI Library

BASIC provides routines which implement software SPI. These routines are hardware independent and can be used with any MCU. You can easily communicate with other devices via SPI - A/D converters, D/A converters, MAX7219, LTC1290 etc. Simply use the following functions and procedures.

5.2.19.1 Soft_SPI_Config – Configure MCU for SPI communication

Prototype	sub procedure Soft_SPI_Config(dim byref Port as byte , const SDI, const SDO, const SCK)
Description	<p>Routine configures and initializes software SPI with the following defaults:</p> <ul style="list-style-type: none"> • Set MCU to master mode, • Clock = 50kHz, • Data sampled at the middle of interval, • Clock idle state low • Data transmitted at low to high edge. <p>SDI pin, SDO pin, and SCK pin are specified by the appropriate parameters.</p>
Example	<pre>Soft_SPI_Config(PORTB, 1, 2, 3) ' SDI pin is RB1, SDO pin is RB2, and SCK pin is RB3.</pre>

5.2.19.2 Soft_SPI_Read – Reads the received data

Prototype	sub function Soft_SPI_read(dim Buffer as byte) as byte
Description	Routine provides clock by sending <Buffer> and reads the received data at the end of the period.

Example	<code>Soft_SPI_Read(dat)</code>
----------------	---------------------------------

5.2.19.3 Soft_SPI_Write – *Sends data via SPI*

Prototype	<code>sub procedure Soft_SPI_Write(dim Data as byte)</code>
Description	Routine writes <i><Data></i> to SSPBUF and immediately starts the transmission.
Example	<code>Soft_SPI_Write(dat)</code>

5.2.20 Software UART Library

BASIC provides routines which implement software UART. These routines are hardware independent and can be used with any MCU. You can easily communicate with other devices via RS232 protocol . Simply use the functions and procedures described below.

5.2.20.1 Soft_UART_Init – *Initializes UART*

Prototype	sub procedure Soft_UART_Init(dim byref Port as byte , const RX, const TX, const Baud_Rate)
Description	Initializes PIC MCU UART at specified pins establishes communication at <i><Baud_Rate></i> . If you specify the unsupported baud rate, compiler will report an error.
Example	Soft_UART_Init(PORTB, 1, 2, 9600)

5.2.20.2 Soft_UART_Read – *Receives a byte*

Prototype	sub function Soft_UART_Read(dim byref Msg_received as byte) as byte
Description	Function returns a received byte. Parameter <i><Msg_received></i> will take true if transfer was succesful. Soft_UART_Read is a non-blocking function call, so you should test <i><Msg_received></i> manually (check the example below).
Example	Received_byte = Soft_UART_Read(Rec_ok)

5.2.20.4 Soft_UART_Write – *Transmits a byte*

Prototype	sub procedure Soft_USART_Write(dim Data as byte)
Description	Procedure transmits byte <i><Data></i> .
Example	Soft_UART_Write(Received_byte)

5.2.21 Sound Library

BASIC provides a sound library which allows you to use sound signalization in your applications.

5.2.21.1 Sound_Init – *Initializes sound engine*

Prototype	sub procedure Sound_Init(dim byref Port, dim Pin as byte)
Description	Procedure Sound_Init initializes sound engine and prepares it for output at specified <i><Port></i> and <i><Pin></i> . Parameter <i><Pin></i> needs to be within range 0..7.
Example	<pre> PORTB = 0 ' Clear PORTB TRISB = 0 ' PORTB is output Sound_Init(PORTB, 2) ' Initialize sound on PORTB.RB2 </pre>

5.2.21.2 Sound_Play – *Plays sound at specified port*

Prototype	sub procedure Sound_Play(dim byref Port, dim Pin as byte)
Description	<p>Procedure Sound_Play plays the sound at the specified port pin. <i><Period_div_10></i> is a sound period given in MCU cycles divided by ten, and generated sound lasts for a specified number of periods (<i><Num_of_Periods></i>).</p> <p>For example, if you want to play sound of 1KHz: $T = 1/f = 1\text{ms} = 1000 \text{ cycles @ } 4\text{MHz}$. code>. This gives us our first parameter: $1000/10 = 100$. Then, we could play 150 periods like this: Sound_Play(100, 150).</p>
Example	<pre>... Sound_Init(PORTB,2) ' Initialize sound on PORTB.RB2 while true adcValue = ADC_Read(2) ' Get lower byte from ADC Sound_Play(adcValue, 200) ' Play the sound wend</pre>

5.2.22 Trigonometry Library

BASIC provides a trigonometry library for applications which involve angle calculations. Trigonometric routines take an angle (in degrees) as parameter of type word and return sine and cosine multiplied by 1000 and rounded up (as integer).

5.2.22.1 SinE3 – Returns sine of angle

Prototype	sub function sinE3(dim Angle as word) as integer
Description	<p>Function takes a word-type number which represents angle in degrees and returns the sine of <i><Angle></i> as integer, multiplied by 1000 (1E3) and rounded up to nearest integer: <code>result = round_up(sin(Angle) * 1000)</code>. Thus, the range of the return values for these functions is from -1000 to 1000.</p> <p>Note that parameter <i><Angle></i> cannot be negative. Function is implemented as lookup table, and the maximum error obtained is ± 1.</p>
Example	<pre> dim angle as word dim result as integer angle = 45 result = sinE3(angle) ' result is 707 </pre>

5.2.22.2 CosE3 – Returns cosine of angle

Prototype	sub function cosE3(dim Angle as word) as integer
Description	<p>Function takes a word-type number which represents angle in degrees and returns the cosine of <i><Angle></i> as integer, multiplied by 1000 (1E3) and rounded up to nearest integer: <code>result = round_up(cos(Angle) * 1000)</code>. Thus, the range of the return values for these functions is from -1000 to 1000.</p> <p>Note that parameter <i><Angle></i> cannot be negative. Function is implemented as lookup table, and the maximum error obtained is ± 1.</p>
Example	<pre> dim angle as word dim result as integer angle = 90 result = cosE3(angle) ' result is 0 </pre>

5.2.23 Utilities

BASIC provides a utility set of procedures and functions for faster development of your applications.

5.2.23.1 Button – *Debounce*

Prototype	sub function Button(dim byref PORT as byte , dim Pin as byte , dim Time as byte , dim Astate as byte) as byte
Description	<p>Function eliminates the influence of contact flickering due to the pressing of a button (debouncing).</p> <p>Parameters <i><PORT></i> and <i><Pin></i> specify the location of the button; parameter <i><Time></i> represents the minimum time interval that pin must be in active state in order to return one; parameter <i><Astate></i> can be only zero or one, and it specifies if button is active on logical zero or logical one.</p>
Example	<pre> if Button(PORTB, 0, 1, 1) then flag = 255 end if </pre>

PIC, PIC, PICmicro, and MPLAB are registered and protected trademarks of the Microchip Technology Inc. USA. Microchip logo and name are the registered tokens of the Microchip Technology. mikroBasic is a registered trade mark of mikroElektronika. All other tokens mentioned in the book are the property of the companies to which they belong.

mikroElektronika © 1998 - 2004. All rights reserved. If you have any questions, please contact our **office**.

Chapter 6: Examples with PIC Integrated Peripherals

- Introduction
- 6.1 Interrupt Mechanism
- 6.2 Internal AD Converter
- 6.3 TMR0 Timer
- 6.4 TMR1 Timer
- 6.5 PWM Module
- 6.6 Hardware UART module (RS-232 Communication)

Introduction

It is commonly said that microcontroller is an “entire computer on a single chip”, which implies that it has more to offer than a single CPU (microprocessor). This additional functionality is actually located in microcontroller’s subsystems, also called the “integrated peripherals”. These (sub)devices basically have two major roles: they expand the possibilities of the MCU making it more versatile, and they take off the burden for some repetitive and “dumber” tasks (mainly communication) from the CPU.

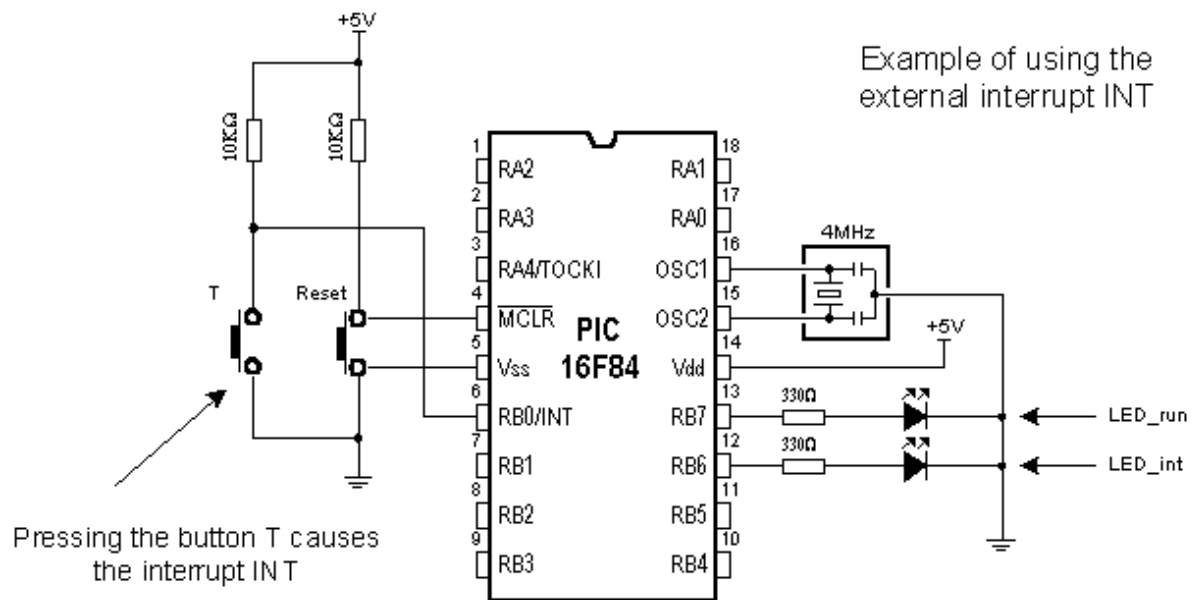
Every microcontroller is supplied with at least a couple of integrated peripherals – commonly, these include timers, interrupt mechanisms and AD converters. More powerful microcontrollers can command a larger number of more diverse peripherals. In this chapter, we will cover some common systems and the ways to utilize them from BASIC programming language.

6.1 Interrupt Mechanism

Interrupts are mechanisms which enable instant response to events such as counter overflow, pin change, data received, etc. In normal mode, microcontroller executes the main program as long as there are no occurrences that would cause an interrupt. Upon interrupt, microcontroller stops the execution of main program and commences the special part of the program which will analyze and handle the interrupt. This part of program is known as the *interrupt (service) routine*.

In BASIC, interrupt service routine is defined by procedure with reserved name `interrupt`. Whatever code is stored in that procedure, it will be executed upon interrupt.

First, we need to determine which event caused the interrupt, as PIC microcontroller calls the same interrupt routine regardless of the trigger. After that comes the interrupt handling, which is executing the appropriate code for the trigger event.



Here is a simple example:

In the main loop, program keeps LED_run diode on and LED_int diode off. Pressing the button T causes the interrupt – microcontroller stops executing the main program and starts the interrupt procedure.

```

program testinterrupt

symbol LED_run = PORTB.7           ' LED_run is connected to PORTB pin
7
symbol LED_int = PORTB.6           ' LED_int is connected to PORTB pin
6

sub procedure interrupt           ' Interrupt service routine

  if INTCON.RBIF = 1 then           ' Changes on RB4-RB7 ?
    INTCON.RBIF = 0

  else if INTCON.INTF = 1 then       ' External interrupt (RB0 pin) ?
    LED_run = 0
    LED_int = 1
    Delay_ms(500)
    INTCON.INTF = 0

    else if INTCON.T0IF = 1 then     ' TMR0 interrupt occurred ?
      INTCON.T0IF = 0

      else if INTCON.EEIF = 1 then   ' Is EEPROM write cycle finished ?
        INTCON.EEIF = 0

  end if

```



```

        end if
    end if
end if
end sub

main:

    TRISB = %00111111      ' Pins RB6 and RB7 are output
    OPTION_REG = %10000000 ' Turn off pull-up resistors
                           ' and set interrupt on falling edge
                           ' of RB0 signal

    INTCON = %10010000     ' Enable external interrupts
    PORTB = 0              ' Initial value on PORTB

    eloop:                 ' While there is no interrupt, program runs in
endless loop:
    LED_run = 1            ' LED_run is on
    LED_int = 0            ' LED_int is off
    goto eloop

end.

```

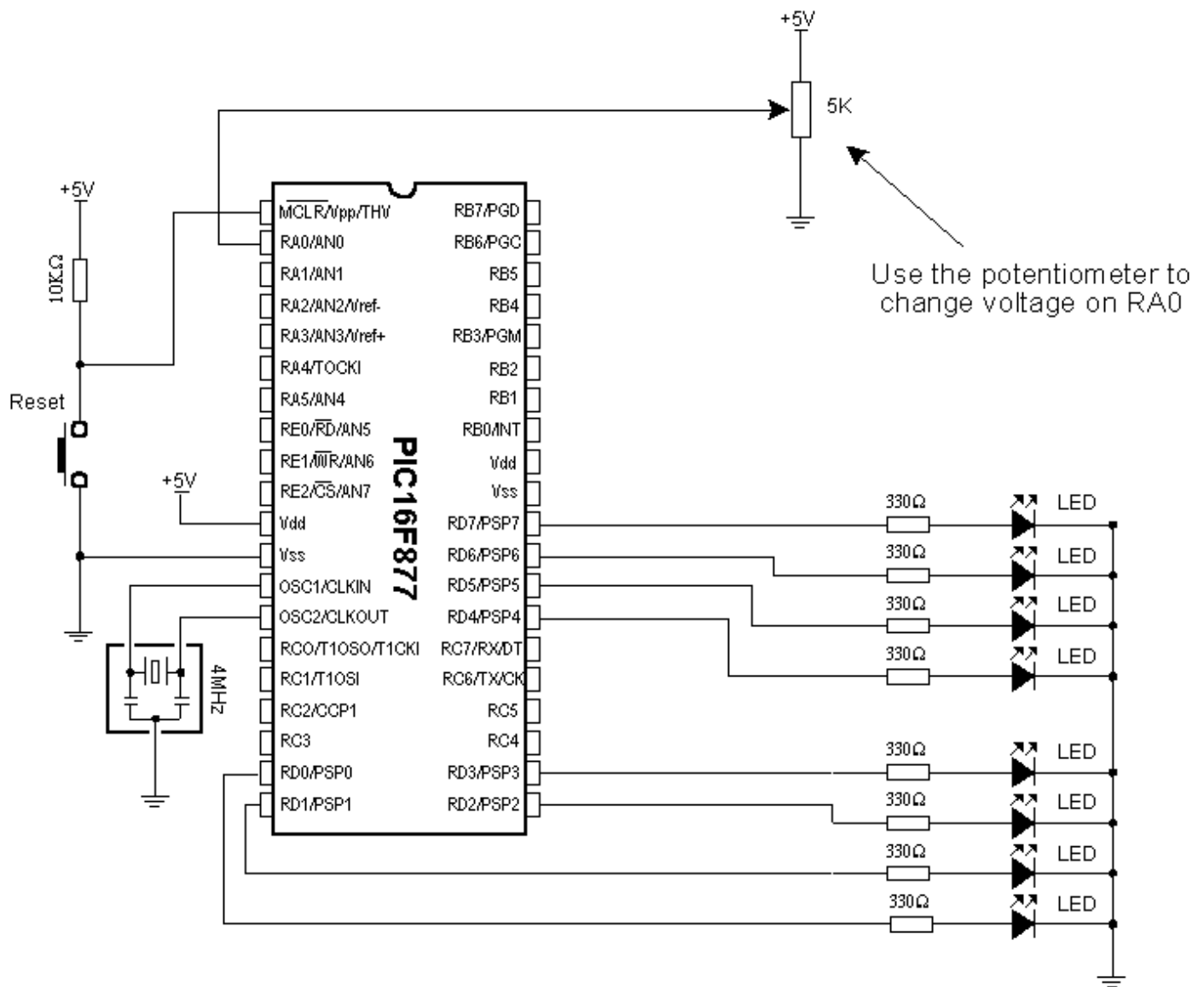
Now, what happens when we push the button? Our interrupt routine first analyzes the interrupt by checking flag bits with couple of `if . . then` instructions, because there are several possible interrupt causes. In our case, an external interrupt took place (pin RB0/INT state changes) and therefore bit INTF in INTCON register is set. Microcontroller will change LED states, and provide a half second delay for us to actually see the change. Then it will clear INTF bit in order to enable interrupts again, and return to executing the main program.

In situations where microcontroller must respond to events unrelated to the main program, it is very useful to have an interrupt service routine. Perhaps, one of the best examples is multiplexing the seven-segment display – if multiplexing code is tied to timer interrupt, main program will be much less burdened because display refreshes in the background.

6.2 Internal AD Converter

A number of microcontrollers have built in Analog to Digital Converter (ADC). Commonly, these AD converters have 8-bit or 10-bit resolution allowing them voltage sensitivity of 19.5mV or 4.8mV, respectively (assuming that default 5V voltage is used).

The simplest AD conversion program would use 8-bit resolution and 5V of microcontroller power as referent voltage (value which the value "read" from the microcontroller pin is compared to). In the following example we measure voltage on RA0 pin which is connected to the potentiometer (see the figure below).



Potentiometer gives 0V in one terminal position and 5V in the other – since we use 8-bit conversion, our digitalized voltage can have 256 steps. The following program reads voltage on RA0 pin and displays it on port B diodes. If not one diode is on, result is zero and if all of diodes are on, result is 255.

```

program ADC_8

main:

TRISA = %111111      ' Port A is input
PORTD = 0
TRISD = %00000000

ADCON1 = %1000010    ' Port A is in analog mode,
                      ' 0 and 5V are referent voltage values,
                      ' and the result is aligned right
                      ' (higher 6 bits of ADRESH are zero).

ADCON0 = %11010001   ' ADC clock is generated by internal RC

```

```

'    circuit; voltage is measured on RA2 and
'    allows the use of AD converter

Delay_ms (500)      ' 500 ms pause

eloop:
    ADCON0.2 = 1      ' Conversion starts

wait:

' wait for ADC to finish
Delay_ms(5)
if ADCON0.2 = 1 then
    goto wait
end if

PORTD = ADRESH      ' Set lower 8 bits on port D
Delay_ms(500)      ' 500 ms pause
goto eloop          ' Repeat all
end.               ' End of program.

```

First, we need to properly initialize registers ADCON1 and ADCON0. After that, we set ADCON0.2 bit which initializes the conversion and then check ADCON0.2 to determine if conversion is over. If over, the result is stored into ADRESH and ADRESL where from it can be copied.

Former example could also be carried out via ADC_Read instruction. Our following example uses 10-bit resolution:

```

program ADC_10

dim AD_Res as word

main:
TRISA  = %11111111      ' PORTA is input

TRISD  = %00000000      ' PORTD is output

ADCON1 = %1000010      ' PORTA is in analog mode,
                        ' 0 and 5V are referent voltage values,
                        ' and the result is aligned right

eloop:
    AD_Res = ADC_read(2) ' Execute conversion and store result

```

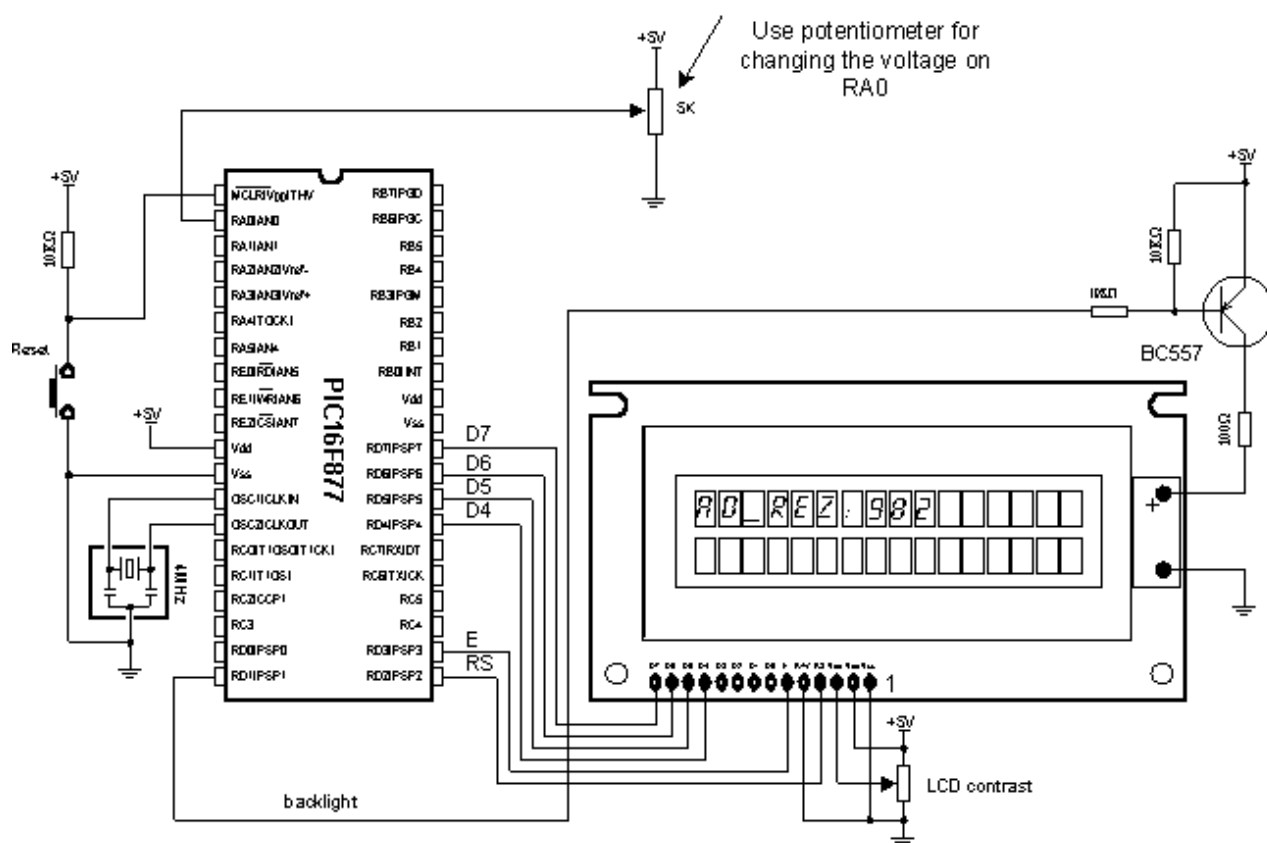
```

'      in variable AD_Res.

PORTD = Lo(AD_Res)      ' Display lower byte of result on PORTD
Delay_ms(500)           ' 500 ms pause
goto eloop              ' Repeat all
end.                    ' End of program

```

As one port is insufficient, we can use LCD for displaying all 10 bits of result. Connection scheme is below and the appropriate program follows. For more information on LCD routines, check Chapter 5.2: Library Routines.



```

program ADC_on_LCD

```

```

dim AD_Res as word

```

```

dim dummyCh as char[6]

```

```

main:

```

```

TRISA  = %1111111      ' PORTA is input

```

```

TRISB  = 0              ' PORTB is output (for LCD)

```

```

ADCON1 = %10000010     ' PORTA is in analog mode,
                        ' 0 and 5V are referent voltage values,

```

```

                                ' and the result is aligned right.

Lcd_Init(PORTB)                ' Initialize LCD
Lcd_Cmd(LCD_CLEAR)             ' Clear LCD
Lcd_Cmd(LCD_CURSOR_OFF)       ' and turn the cursor off

eloop:

AD_Res = ADC_Read(2)           ' Execute conversion and store result
                                ' to variable AD_Res

LCD_Out(1, 1, "                ") ' Clear LCD from previous result
WordToStr(AD_Res, dummyCh)     ' Convert the result in text,
LCD_Out(1, 1, dummyCh)         ' and print it in line 1, char 1

Delay_ms(500)                  ' 500 ms pause
goto eloop                     ' Repeat all
end.                           ' End of program

```

6.3 TMR0 Timer

TMR0 timer is an 8-bit special function register with working range of 256. Assuming that 4MHz oscillator is used, TMR0 can measure 0-255 microseconds range (at 4MHz, TMR0 increments by one microsecond). This period can be increased if prescaler is used. Prescaler divides clock in a certain ratio (prescaler settings are made in OPTION_REG register).

Our following program example shows how to generate 1 second using TMR0 timer. For visual purposes, program toggles LEDs on PORTB every second.

Before the main program, TMR0 should have interrupt enabled (bit 2) and GIE bit (bit 7) in INTCON register should be set. This will enable global interrupts.

```

program Timer0_1sec

dim cnt as byte
dim a as byte
dim b as byte

sub procedure interrupt
    cnt = cnt + 1                ' Increment value of cnt on every interrupt
    TMR0 = 96
    INTCON = $20                ' Set T0IE, clear T0IF
end sub

```

```

main:

a = 0
b = 1
OPTION_REG = $84      ' Assign prescaler to TMR0
TRISB  = 0            ' PORTB as output
PORTB  = $FF          ' Initialize PORTB
cnt = 0               ' Initialize cnt
TMR0   = 96
INTCON = $A0          ' Enable TMR0 interrupt

' If cnt is 200, then toggle PORTB LEDs and reset cnt
do
  if cnt = 200 then
    PORTB = not(PORTB)
    cnt = 0
  end if
loop until 0 = 1

end.

```

Prescaler is set to 32, so that internal clock is divided by 32 and TMR0 increments every 31 microseconds. If TMR0 is initialized at 96, overflow occurs in $(256-96)*31 \text{ us} = 5 \text{ ms}$. We increase *cnt* every time interrupt takes place, effectively measuring time according to the value of this variable. When *cnt* reaches 200, time will total $200*5 \text{ ms} = 1 \text{ second}$.

6.4 TMR1 Timer

TMR1 timer is a 16-bit special function register with working range of 65536. Assuming that 4MHz oscillator is used, TMR1 can measure 0-65535 microseconds range (at 4MHz, TMR1 increments by one microsecond). This period can be increased if prescaler is used. Prescaler divides clock in a certain ratio (prescaler settings are made in T1CON register).

Before the main program, TMR1 should be enabled by setting the zero bit in T1CON register. First bit of the register defines the internal clock for TMR1 – we set it to zero. Other important registers for working with TMR1 are PIR1 and PIE1. The first contains overflow flag (zero bit) and the other is used to enable TMR1 interrupt (zero bit). With TMR1 interrupt enabled and its flag cleared, we only need to enable global interrupts and peripheral interrupts in the INTCON register (bits 7 and 6, respectively).

Our following program example shows how to generate 10 seconds using TMR1 timer. For visual purposes, program toggles LEDs on PORTB every 10 seconds.

```

program Timer1_10sec

```

```

dim cnt as byte

sub procedure interrupt
    cnt = cnt + 1
    pir1.0 = 0      ' Clear TMR1IF
end sub

main:

TRISB = 0
T1CON = 1
PIR1.TMR1IF = 0    ' Clear TMR1IF
PIE1 = 1           ' Enable interrupts
PORTB = $F0
cnt = 0            ' Initialize cnt
INTCON = $C0

' If cnt is 152, then toggle PORTB LEDs and reset cnt
do
    if cnt = 152 then
        PORTB = not(PORTB)
        cnt = 0
    end if
loop until 0 = 1

end.

```

Prescaler is set to 00 so there is no dividing the internal clock and overflow occurs every 65.536 ms. We increase *cnt* every time interrupt takes place, effectively measuring time according to the value of this variable. When *cnt* reaches 152, time will total $152 \times 65.536 \text{ ms} = 9.96 \text{ seconds}$.

6.5 PWM Module

Microcontrollers of PIC16F87X series have one or two built-in PWM outputs (40-pin casing allows 2, 28-pin casing allows 1). PWM outputs are located on RC1 and RC2 pins (40-pin MCUs), or on RC2 pin (28-pin MCUs). Refer to PWM library (Chapter 5.2: Library Routines) for more information.

The following example uses PWM library for getting various light intensities on LED connected to RC2 pin. Variable which represents the ratio of on to off signals is continually increased in the loop, taking values from 0 to 255. This results in continual intensifying of light on LED diode. After value of 255 has been reached, process begins anew.

```

program PWM_LED_Test

```

```

dim j as byte

main:

TRISB = 0           ' PORTB is output
PORTB = 0           ' Set PORTB to 0
j      = 0
TRISC = 0           ' PORTC is output
PORTC = $FF         ' Set PORTC to $FF
PWM_Init(5000)      ' Initialize PWM module
PWM_Start           ' Start PWM

while true         ' Endless loop
    Delay_ms(10)     ' Wait 10ms
    j = j + 1       ' Increment j
    PWM_Change_Duty(j) ' Set new duty ratio
    PORTB = CCPR1L   ' Send value of CCPR1L to PORTB
wend

end.

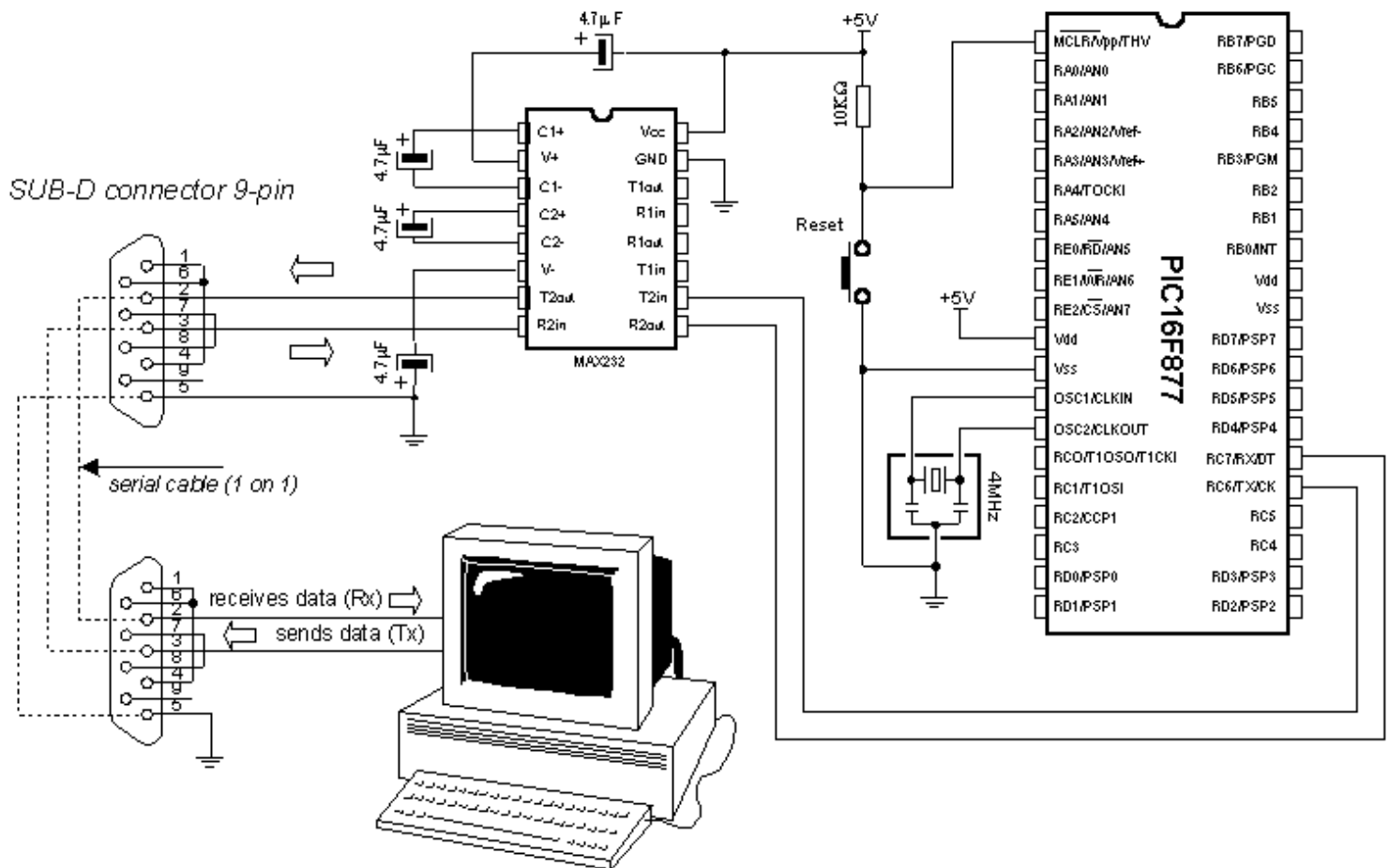
```

6.6 Hardware UART module (RS-232 Communication)

The easiest way to transfer data between microcontroller and some other device, e.g. PC or other microcontroller, is the RS-232 communication (also referred to as EIA RS-232C or V.24). RS232 is a standard for serial binary data interchange between a DTE (Data terminal equipment) and a DCE (Data communication equipment), commonly used in personal computer serial ports. It is a serial asynchronous 2-line (Tx for transmitting and Rx for receiving) communication with effective range of 10 meters.

Microcontroller can establish communication with serial RS-232 line via hardware UART (Universal Asynchronous Receiver Transmitter) which is an integral part of PIC16F87X microcontrollers. UART contains special buffer registers for receiving and transmitting data as well as a Baud Rate generator for setting the transfer rate.

This example shows data transfer between the microcontroller and PC connected by RS-232 line interface MAX232 which has role of adjusting signal levels on the microcontroller side (it converts RS-232 voltage levels +/- 10V to TTL levels 0-5V and vice versa).



Our following program example illustrates use of hardware serial communication. Data received from PC is stored into variable *dat* and sent back to PC as confirmation of successful transfer. Thus, it is easy to check if communication works properly. Transfer format is 8N1 and transfer rate is 2400 baud.

```

program USART_Echo

dim dat as byte

main:

USART_Init(2400)           ' Initialize USART module
while true
    if USART_Data_Ready = 1 then    ' If data is received
        dat = USART_Read            ' Read the received data
        USART_Write(dat)            ' Send data via USART
    end if
wend

end.

```

In order to establish the communication, PC must have a communication software installed. One such communication terminal is part of mikroBasic IDE. It can be accessed by clicking Tools > Terminal from the drop-down menu. Terminal

allows you to monitor transfer and to set all the necessary transfer settings. First of all, we need to set the transfer rate to 2400 to match the microcontroller's rate. Then, select the appropriate communication port by clicking one of the 4 available (check where you plugged the serial cable).

After making these adjustments, clicking Connect starts the communication. Type your message and click Send Message – message will be sent to the microcontroller and back, where it will be displayed on the screen.

Note that serial communication can also be software based on any of 2 microcontroller pins – for more information, check the Chapter 9: Communications.

PIC, PIC, PICmicro, and MPLAB are registered and protected trademarks of the Microchip Technology Inc. USA. Microchip logo and name are the registered tokens of the Microchip Technology. mikroBasic is a registered trade mark of mikroElektronika. All other tokens mentioned in the book are the property of the companies to which they belong.

mikroElektronika © 1998 - 2004. All rights reserved. If you have any questions, please contact our **office**.

Chapter 7: Examples with Displaying Data

- Introduction
- 7.1 LED Diode
- 7.2 Seven-Segment Display
- 7.3 LCD Display, 4-bit and 8-bit Interface
- 7.4 Graphical LCD
- 7.5 Sound Signalization

Introduction

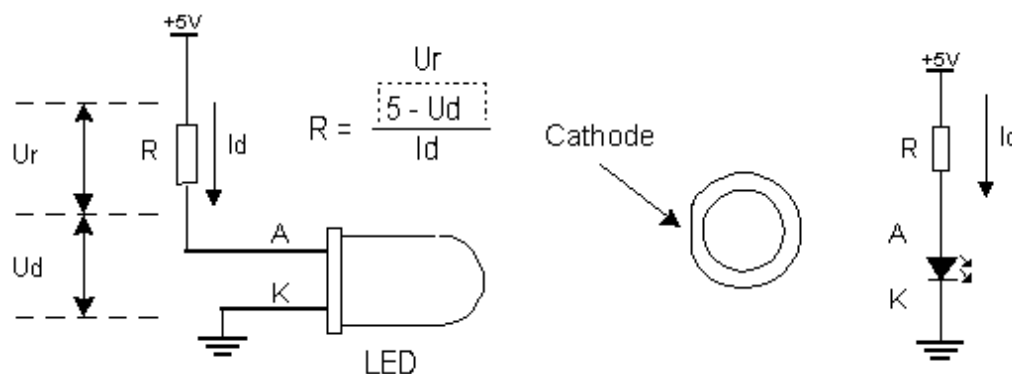
Microcontrollers deal very well with 0's and 1's, but humans do not. We need indicator lights, numbers, letters, charts, beepers... In order to comprehend the information presented quicker and better, we need that information to be displayed to us in many different ways. In practice, human - machine communication can require substantial (machine) resources, so it is sometimes better to dedicate an entire microcontroller to that task. This device is then called the Human - Machine Interface or simply HMI. The second microcontroller is then required to get the human wishes from HMI, "do the job" and put the results back to HMI, so that operator can see it.

Clearly, the most important form of communication between the microcontroller system and a man is the visual communication. In this chapter we will discuss various ways of displaying data, from the simplest to more elaborate ones. You'll see how to use LED diodes, Seven-Segment Displays, character- and graphic LCDs. We will also consider using BASIC for sound signalization necessary in certain applications.

Just remember: the more profound communication you wish to be, the more MCU resources it'll take.

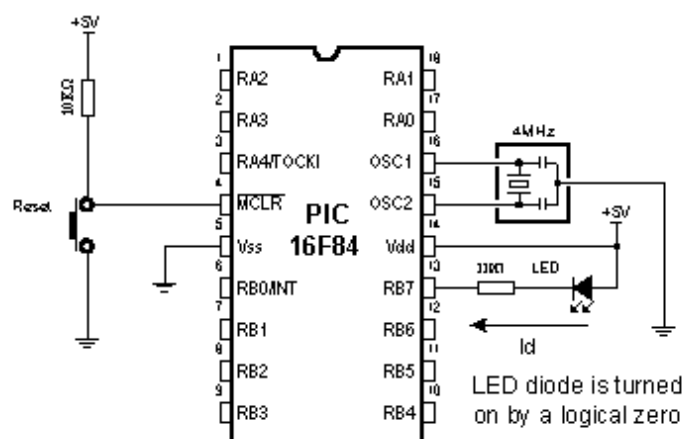
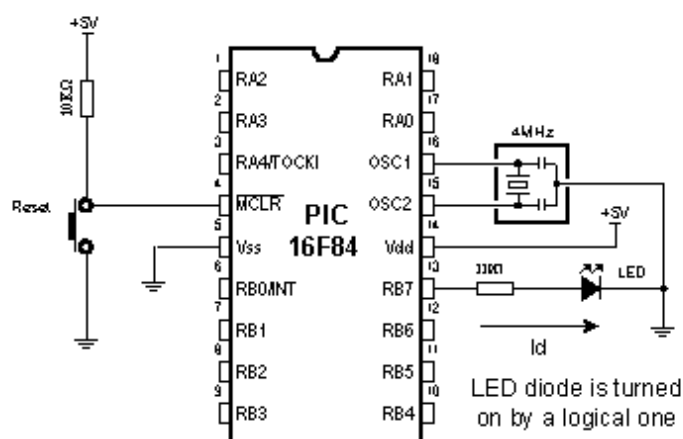
7.1 LED Diode

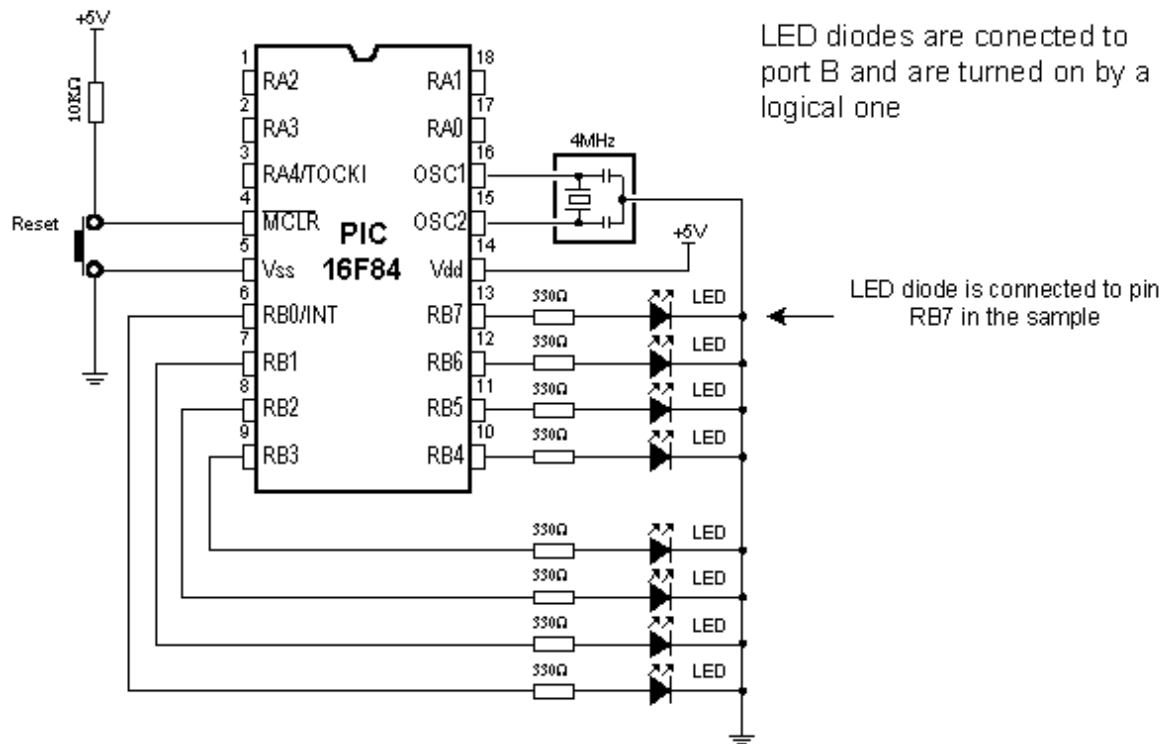
One of the most frequently used components in electronics is surely the LED diode (LED stands for Light Emitting Diode). Some of common LED diode features include: size, shape, color, working voltage (Diode voltage) U_d and electric current I_d . LED diode can be round, rectangular or triangular in shape, although manufacturers of these components can produce any shape needed for specific purposes. Size i.e. diameter of round LED diodes ranges from 3 to 12 mm, with 3 - 5 mm sizes most commonly used. Common colors include red, yellow, green, orange, blue, etc. Working voltage is 1.7V for red, 2.1V for green and 2.3 for orange color. This voltage can be higher depending on the manufacturer. Normal current I_d through diode is 10 mA, while maximal current reaches 25 mA. High current consumption can present problem to devices with battery power supply, so in that case low current LED diode ($I_d \sim 1-2$ mA) should be used. For LED diode to emit light with maximum capacity, it is necessary to connect it properly or it might get damaged.



The positive pole is connected to anode, while ground is connected to cathode. For matter of differentiating the two, cathode is marked by mark on casing and shorter pin. Diode will emit light only if current flows from anode to cathode; in the other case there will be no current. Resistor is added serial to LED diode, limiting the maximal current through diode and protecting it from damage. Resistor value can be calculated from the equation on the picture above, where U_r represents voltage on resistor. For +5V power supply and 10 mA current resistor used should have value of 330•.

LED diode can be connected to microcontroller in two ways. One way is to have microcontroller "turning on" LED diode with logical one and the other way is with logical zero. The first way is not so frequent (which doesn't mean it doesn't have applications) because it requires the microcontroller to be diode current source. The second way works with higher current LED diodes.





The following example toggles LEDs of PORTB every second.

```

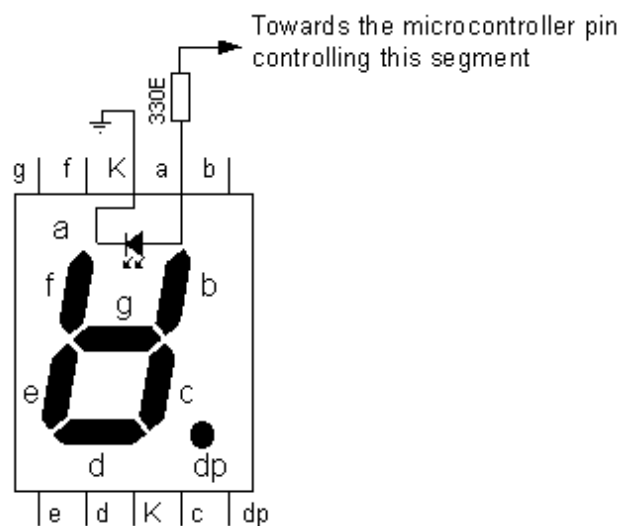
program LED_Blinking

main:
    TRISB = 0           ' PORTB is output
    PORTB = %11111111   ' Turn ON diodes on PORTB
    Delay_ms(1000)      ' Wait for 1 second
    PORTB = %00000000   ' Turn OFF diodes on PORTB
    Delay_ms(1000)      ' Wait for 1 second
    goto main          ' Endless loop
end.

```

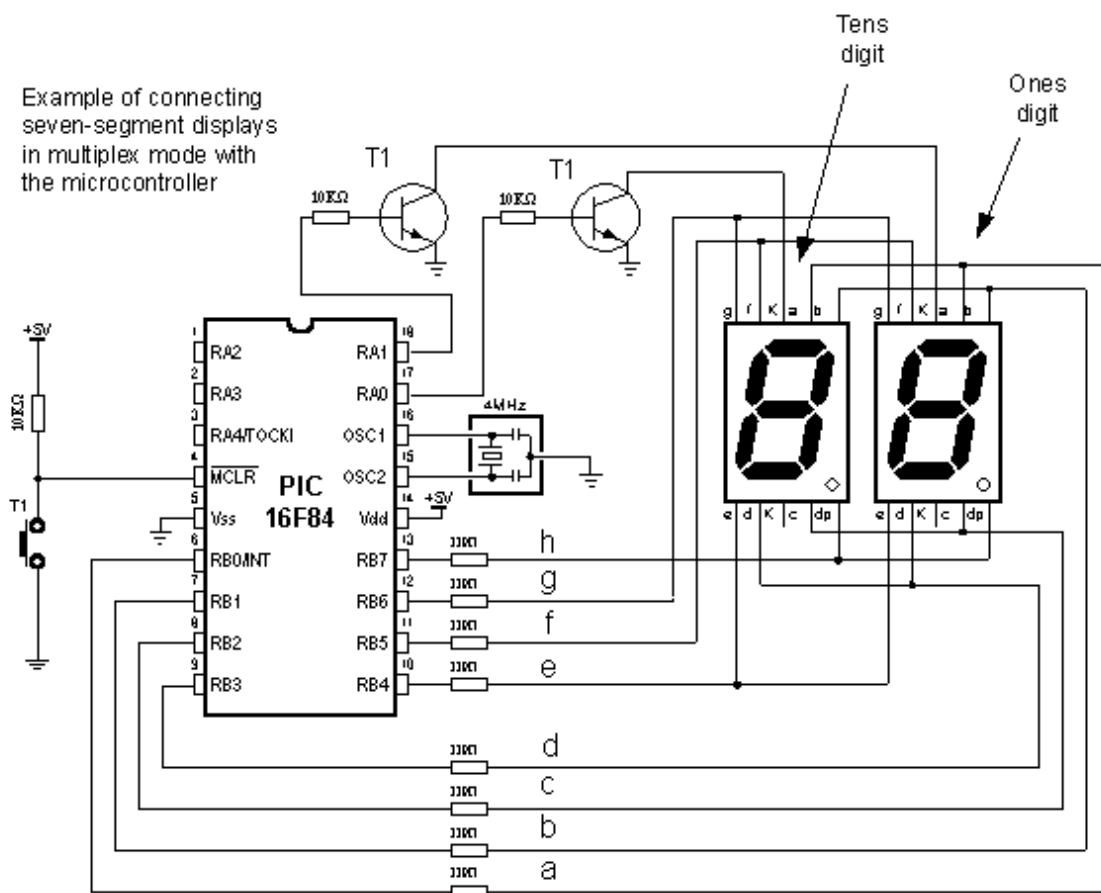
7.2 Seven-Segment Displays

Seven-segment digits represent more advanced form of visual communication. The name comes from the seven diodes (there is an eighth diode for a dot) arranged to form decimal digits from 0 to 9. Appearance of a seven-segment digit is given on a picture below.



As seven-segment digits have better temperature tolerance and visibility than LCD displays, they are very common in industrial applications. Their use satisfies all criteria including the financial one. They are commonly used for displaying value read from sensors, etc.

One of the ways to connect seven-segment display to the microcontroller is given in the figure below. System is connected to use seven-segment digits with common cathode. This means that segments emit light when logical one is brought to them, and that output of all segments must be a transistor connected to common cathode, as shown on the picture. If transistor is in conducting mode any segment with logical one will emit light, and if not no segment will emit light, regardless of its pin state.



Bases of transistors T1 and T2 are connected to pin0 and pin1 of PORTA. Setting those pins turns on the transistor, allowing every segment from "a" to "h", with logical one on it, to emit light. If zero is on transistor base, none of the segments will emit light, regardless of the pin state.

Using the previous scheme, we could display a sequence of nine digits like this:

```

program seven_seg_onedigit

dim i as byte

' Function mask returns mask of parameter 'num'
' for common cathode 7-seg. display

sub function mask(dim num as byte) as byte

    select case num
        case 0 result = $3F
        case 1 result = $06
        case 2 result = $5B
        case 3 result = $4F
        case 4 result = $66
        case 5 result = $6D
        case 6 result = $7D
        case 7 result = $07
        case 8 result = $7F
        case 9 result = $6F
    end select
end sub

main:

INTCON = 0          ' Disable PEIE, INTE, RBIE, TOIE
TRISA  = 0
TRISB  = 0
PORTB  = 0
PORTA  = 2

do
    for i = 0 to 9
        PORTB = mask(i)
        Delay_ms(1000)
    next i
loop until false ' Endless loop

end.

```

Purpose of the program is to display numbers 0 to 9 on the ones digit, with 1 second delay. In order to display a number,

its mask must be sent to PORTB. For example, if we need to display "1", segments b and c must be set to 1 and the rest must be zero. If (according to the scheme above) segments b and c are connected to the first and the second pin of PORTB, values 0000 and 0110 should be set to PORTB. Thus, mask for number "1" is value 0000 0110 or 06 hexadecimal. The following table contains corresponding mask values for numbers 0-9:

Digit	Seg. h	Seg. g	Seg. f	Seg. e	Seg. d	Seg. c	Seg. b	Seg. a	HEX
0	0	0	1	1	1	1	1	1	\$3F
1	0	0	0	0	0	1	1	0	\$06
2	0	1	0	1	1	0	1	1	\$5B
3	0	1	0	0	1	1	1	1	\$4F
4	0	1	1	0	0	1	1	0	\$66
5	0	1	1	0	1	1	0	1	\$6D
6	0	1	1	1	1	1	0	1	\$7D
7	0	0	0	0	0	1	1	1	\$07
8	0	1	1	1	1	1	1	1	\$7F
9	0	1	1	0	1	1	1	1	\$6F

You are not, however, limited to displaying digits. You can use 7seg Display Decoder, a built-in tool of mikroBasic, to get hex code of any other viable combination of segments you would like to display.

But what do we do when we need to display more than one digit on two or more displays? We have to put a mask on one digit quickly enough and activate its transistor, then put the second mask and activate the second transistor (of course, if one of the transistors is in conducting mode, the other should not work because both digits will display the same value). The process is known as “multiplexing”: digits are displayed in a way that human eye gets impression of simultaneous display of both digits – actually only one display emits at any given moment.

Now, let’s say we need to display number 38. First, the number should be separated into tens and ones (in this case, digits 3 and 8) and their masks sent to PORTB. The rest of the program is very similar to the last example, except for having one transition caused by displaying one digit after another:

```

program seven_seg_twodigits

dim    v      as byte
dim    por1 as byte
dim    por2 as byte

sub procedure interrupt
begin
  if v = 0 then
    PORTB = por2      ' Send mask of tens to PORTB

```



```

    PORTA = 1      ' Turn on 1st 7seg, turn off 2nd
    v = 1
else
    PORTB = por1   ' Send mask of ones to PORTB
    PORTA = 2      ' Turn on 2nd 7seg, turn off 1st
    v = 0
end if
TMR0 = 0          ' Clear TMRO
INTCON = $20      ' Clear TMR0IF and set TMR0IE
end sub

main:

OPTION_REG = $80   ' Pull-up resistors
TRISA      = 0     ' PORTA is output
TRISB      = 0     ' PORTB is output
PORTB      = 0     ' Clear PORTB (make sure LEDs are off)
PORTA      = 0     ' Clear PORTA (make sure both displays are off)
TMR0       = 0     ' Clear TMRO
por1       = $7F   ' Mask for '8' (check the table above)
por2       = $4F   ' Mask for '3' (check the table above)
INTCON     = $A0   ' Enable T0IE

while true      ' Endless loop, wait for interrupt
    nop
wend

end.

```

The multiplexing problem is solved for now, but your program probably doesn't have a sole purpose of printing constant values on 7seg display. It is usually just a subroutine for displaying certain information. However, this approach to printing data on display has proven to be very convenient for more complicated programs. You can also move part of the program for refreshing the digits (handling the masks) to the interrupt routine.

The following example increases variable *i* from 0 to 99 and prints it on displays. After reaching 99, counter begins anew.

```

program seven_seg_counting

dim    i    as byte
dim    j    as byte
dim    v    as byte
dim    por1 as byte

```

```
dim      por2 as byte
```

```
' This function returns masks
' for common cathode 7-seg display
```

```
sub function mask(dim num as byte) as byte
```

```
    select case num
        case 0 result = $3F
        case 1 result = $06
        case 2 result = $5B
        case 3 result = $4F
        case 4 result = $66
        case 5 result = $6D
        case 6 result = $7D
        case 7 result = $07
        case 8 result = $7F
        case 9 result = $6F
    end select
```

```
end sub
```

```
sub procedure interrupt
```

```
    if v = 0 then
        PORTB = por2      ' Prepare mask for digit
        PORTA = 1         ' Turn on 1st, turn off 2nd 7seg
        v = 1
    else
        PORTB = por1      ' Prepare mask for digit
        PORTA = 2         ' Turn on 2nd, turn off 1st 7seg
        v = 0
    end if
    TMR0 = 0
    INTCON = $20
```

```
end sub
```

```
main:
```

```
OPTION_REG = $80
por2       = $3F
j          = 0
TMR0       = 0
INTCON     = $A0      ' Disable PEIE, INTE, RBIE, TOIE
TRISA     = 0
```

```

TRISB      =    0
PORTB      =    0
PORTA      =    0

do
  for i = 0 to 99      ' Count from 0 to 99

    ' Prepare ones digit
    j = i mod 10
    por1 = mask(j)

    ' Prepare tens digit
    j = (i div 10) mod 10
    por2 = mask(j)

    Delay_ms(1000)

  next i
loop until false

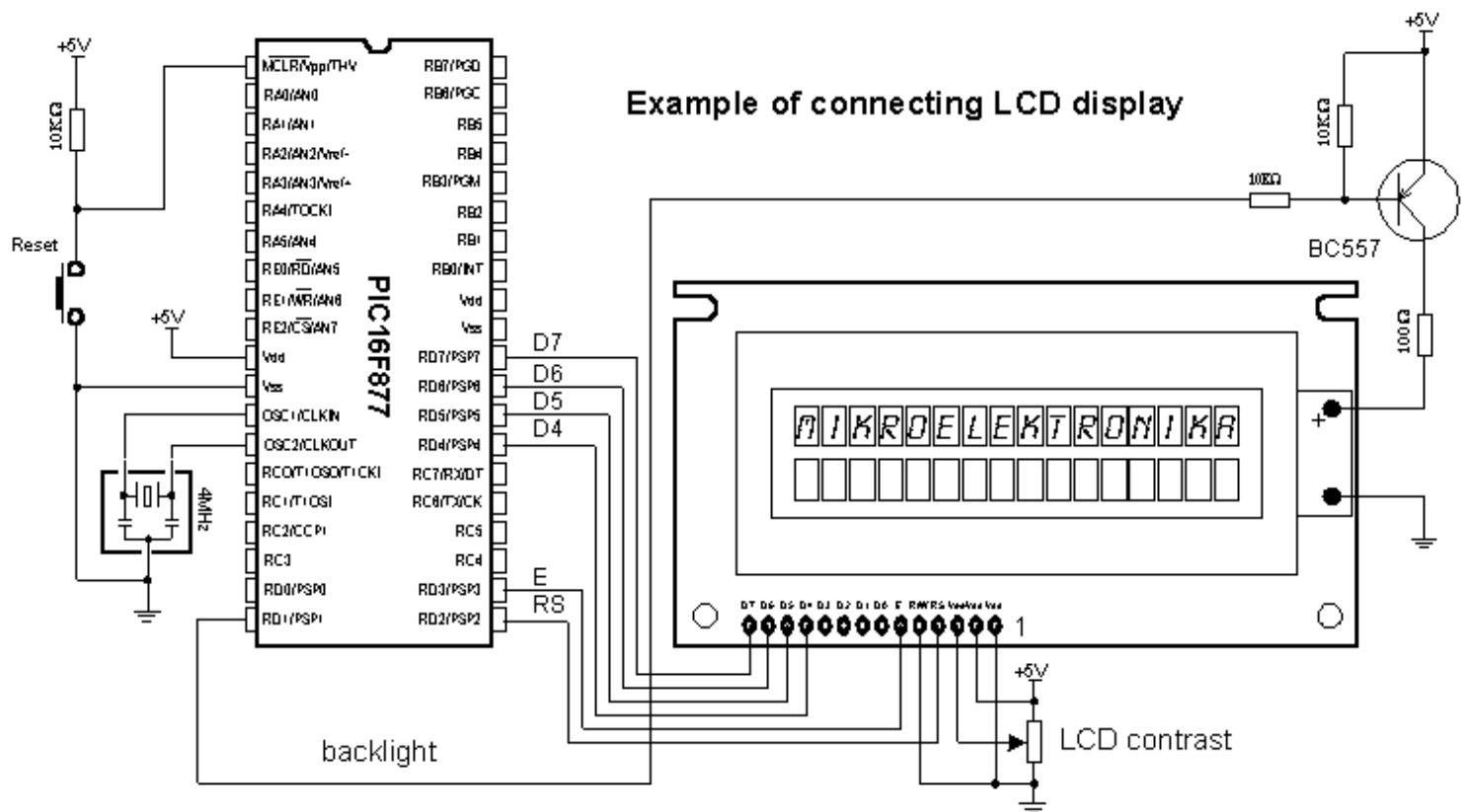
end.

```

In the course of the main program, programmer doesn't need to worry of refreshing the display. Just call the subroutine *mask* every time display needs to change.

7.3 LCD Display, 4-bit and 8-bit Interface

One of the best solutions for devices that require visualizing the data is the “smart” Liquid Crystal Display (LCD). This type of display consists of 7x5 dot segments arranged in rows. One row can consist of 8, 16, 20, or 40 segments, and LCD display can have 1, 2, or 4 rows.



LCD connects to microcontroller via 4-bit or 8-bit bus (4 or 8 lines). R/W signal is on the ground, because communication is one-way (toward LCD). Some displays have built-in backlight that can be turned on with RD1 pin via PNP transistor BC557.

Our following example prints text on LCD via 4-bit interface. Assumed pin configuration is default.

```

program LCD_default_test

dim Text as char[20]

main:

TRISB = 0                                ' PORTB is output
LCD_Init(PORTB)                          ' Initialize LCD at PORTB
LCD_Cmd(LCD_CURSOR_OFF)                  ' Turn off cursor
Text = "mikroelektronika"
LCD_Out(1, 1, Text)                       ' Print text at LCD

end.

```

Our second example prints text on LCD via 8-bit interface, with custom pin configuration.

```

program LCD8_test

```

```

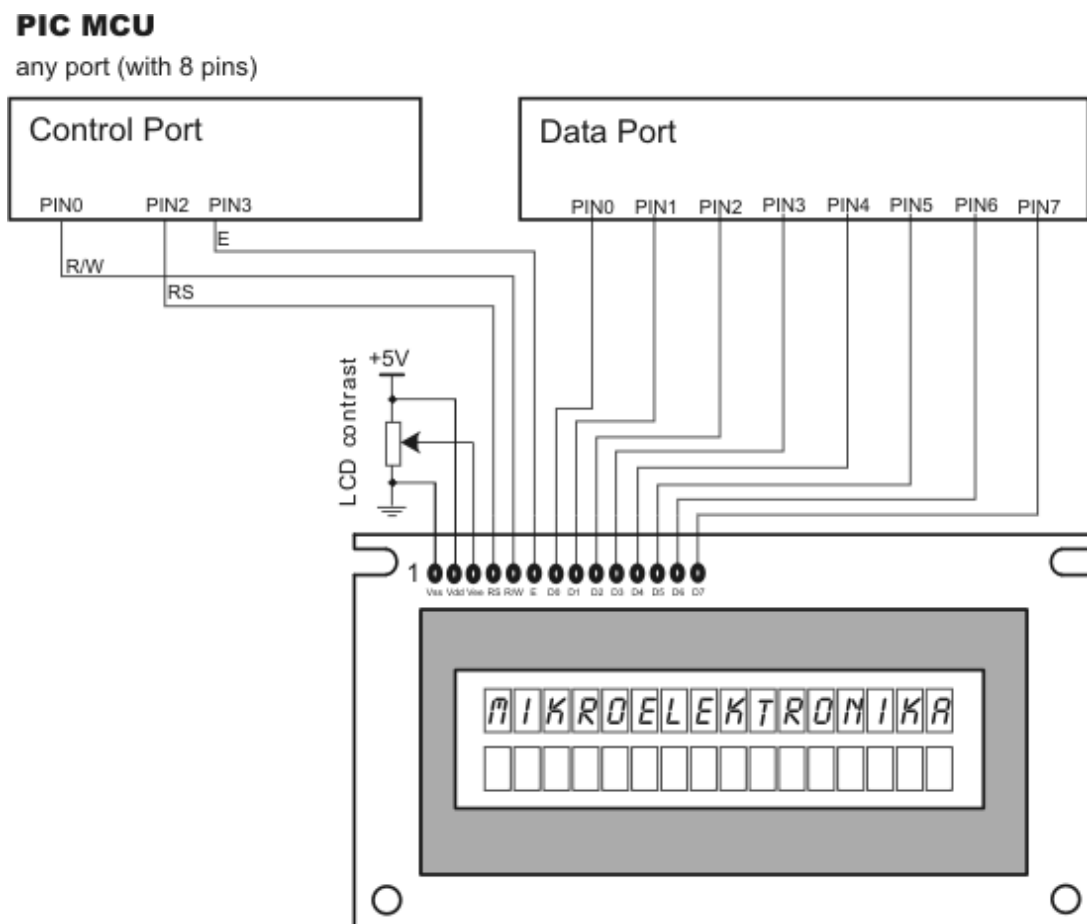
dim Text as char[20]

main:
    TRISB = 0                                ' PORTB is output
    TRISD = 0                                ' PORTD is output

    ' Initialize LCD at PORTB and PORTD with custom pin settings
    LCD8_Config(PORTB,PORTD,2,3,0,7,6,5,4,3,2,1,0)

    LCD8_Cmd(LCD_CURSOR_OFF)                ' Turn off cursor
    Text = "mikroElektronika"
    LCD8_Out(1, 1, Text)                    ' Print text at LCD
end.

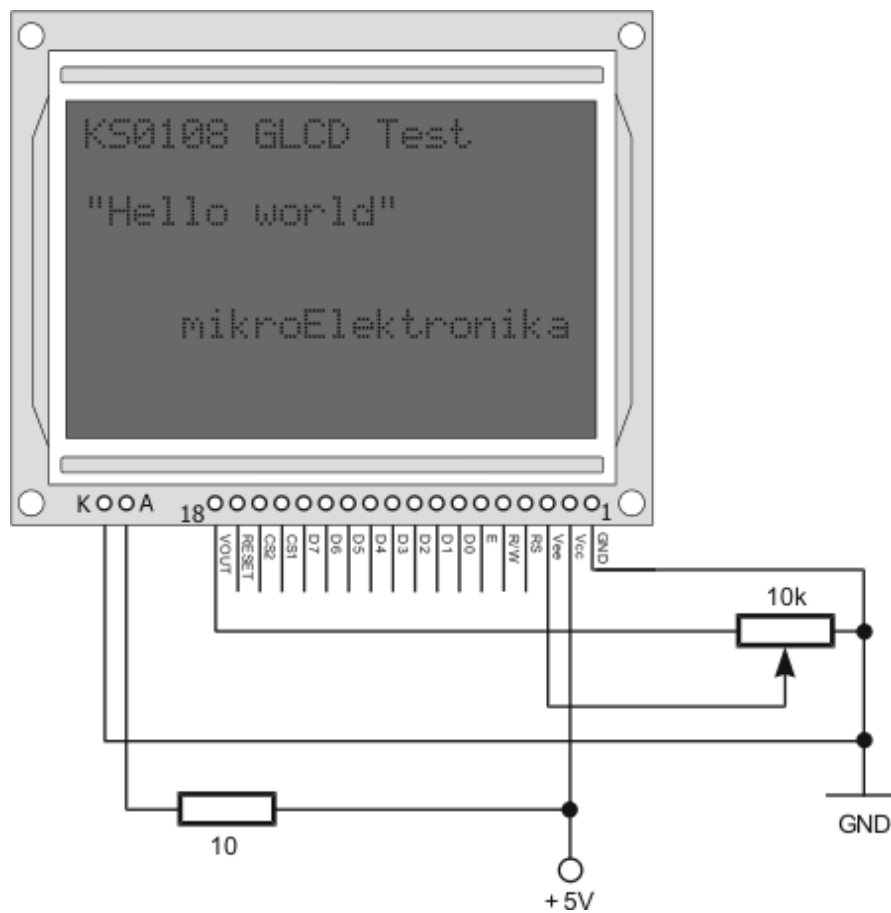
```



7.4 Graphical LCD (PIC18 only)

Most commonly used Graphical LCD (GLCD) has screen resolution of 128x64 pixels. This allows creating more elaborate visual messages than usual LCD can provide, involving drawings and bitmaps.

The following figure shows GLCD HW connection by default initialization (using GLCD_LCD_Init routine); if you need different pin settings, refer to GLCD_LCD_Config.



BASIC offers a comprehensive library for GLCD – refer to Chapter 5: Built-in and Library Routines for more information. Our following example demonstrates the possibilities of GLCD and the mentioned library. Note that the library works with PIC18 only.

```

program GLCD_test
  ' For PIC18

include "GLCD_128x64.pbas"      ' You need to include GLCD_128x64 library

dim text as string[25]

main:

  PORTC = 0
  PORTB = 0
  PORTD = 0
  TRISC = 0
  TRISD = 0
  TRISB = 0

  GLCD_LCD_Init(PORTC, PORTD)    ' default settings
  GLCD_Set_Font(FONT_NORMAL1)

```

```

while true
    GLCD_Clear_Screen

    ' Draw Circles
    GLCD_Clear_Screen
    text = "Circle"
    GLCD_Put_Text(0, 7, text, NONINVERTED_TEXT)
    GLCD_Circle(63,31,10)

    Delay_Ms(4000)

    ' Draw Rectangles
    GLCD_Clear_Screen
    text = "Rectangle"
    GLCD_Put_Text(0, 7, text, NONINVERTED_TEXT)

    GLCD_Rectangle(10, 0, 30, 35)

    Delay_Ms(4000)
    GLCD_Clear_Screen

    ' Draw Lines
    GLCD_Clear_Screen
    text = "Line"
    GLCD_Put_Text(55, 7, text, NONINVERTED_TEXT)

    GLCD_Line(0, 0, 127, 50)

    GLCD_Line(0,63, 50, 0)

    Delay_Ms(5000)

    ' Fonts Demo
    GLCD_Clear_Screen
    text = "Fonts DEMO"
    GLCD_Set_Font(FONT_TINY)
    GLCD_Put_Text(0, 4, text, NONINVERTED_TEXT)
    GLCD_Put_Text(0, 5, text, INVERTED_TEXT)
    GLCD_Set_Font(FONT_BIG)
    GLCD_Put_Text(0, 6, text, NONINVERTED_TEXT)
    GLCD_Put_Text(0, 7, text, INVERTED_TEXT)
    Delay_ms(5000)

wend

end.

```

7.5 Sound Signalization

Some applications require sound signalization in addition to visual or instead of it. It is commonly used to alert or announce the termination of some long, time-consuming process. The information presented by such means is fairly simple, but relieves the user from having to constantly look into displays and dials.

BASIC's Sound library facilitates generating sound signals and output on specified port. We will present a simple demonstration using piezzo speaker connected to microcontroller's port.

```

program Sound

' The following three tones are calculated for 4MHz crystal
sub procedure Tone1
    Sound_Play(200, 200)      ' Period = 2ms <=> 500Hz, Duration = 200 periods
end sub

sub procedure Tone2
    Sound_Play(180, 200)      ' Period = 1.8ms <=> 555Hz
end sub

sub procedure Tone3
    Sound_Play(160, 200)      ' Period = 1.6ms <=> 625Hz
end sub

sub procedure Melody          ' Plays the melody "Yellow house"

    Tone1
    Tone2
    Tone3
    Tone3

    Tone1
    Tone2
    Tone3
    Tone3

    Tone1
    Tone2
    Tone3

    Tone1
    Tone2
    Tone3
    Tone3
  
```



```

Tone1
Tone2
Tone3

Tone3
Tone3
Tone2
Tone1
end sub

main:
    TRISB = $F0

    Sound_Init(PORTB, 2)           ' Connect speaker on pins RB2 and
    GND                             GND
    Sound_Play(50, 100)

    while true
        if Button(PORTB,7,1,1) then           ' RB7 plays Tone1
            Tone1
        end if
        while TestBit(PORTB,7) = 1           ' Wait for button to be released
            nop
        wend

        if Button(PORTB,6,1,1) then           ' RB6 plays Tone2
            Tone2
        end if
        while TestBit(PORTB,6) = 1           ' Wait for button to be released
            nop
        wend

        if Button(PORTB,5,1,1) then           ' RB5 plays Tone3
            Tone3
        end if
        while TestBit(PORTB,5) = 1           ' Wait for button to be released
            nop
        wend

        if Button(PORTB,4,1,1) then           ' RB4 plays Melody
            Melody
        end if
        while TestBit(PORTB,4) = 1           ' Wait for button to be released
            nop

```

```
wend
```

```
wend
```

```
end.
```

PIC, PIC, PICmicro, and MPLAB are registered and protected trademarks of the Microchip Technology Inc. USA. Microchip logo and name are the registered tokens of the Microchip Technology. mikroBasic is a registered trade mark of mikroElektronika. All other tokens mentioned in the book are the property of the companies to which they belong.

mikroElektronika © 1998 - 2004. All rights reserved. If you have any questions, please contact our **office**.

Chapter 8: Examples with Memory and Storage Media

- Introduction
- 8.1 EEPROM Memory
- 8.2 Flash Memory
- 8.3 Compact Flash

Introduction

There is no program on this world that doesn't interact with memory in some way. First, during its execution, it retains the operational data from, uses or alters it, and puts it back into the program memory. Second, it is often necessary to store and handle large amount of data that can be obtained from various sources, whether it is the car engine temperature acquisition data or some bitmap image to be displayed on the GLCD. In this chapter we will focus on the latter problem, i.e. we'll go through the techniques of manipulating data on the so-called memory storage devices and systems.

8.1 EEPROM Memory

Data used by microcontroller is stored in the RAM memory as long as there is a power supply present. If we need to keep the data for later use, it has to be stored in a permanent memory. An EEPROM (E²PROM), or Electrically-Erasable Programmable Read-Only Memory is a non-volatile storage chip, commonly used with PIC microcontrollers for this purpose. An EEPROM can be programmed and erased multiple times electrically – it may be erased and reprogrammed only a certain number of times, ranging from 100,000 to 1,000,000, but it can be read an unlimited number of times.

8.1.1 Internal EEPROM

Some PIC microcontrollers have internal EEPROM allowing you to store information without any

external hardware.

BASIC has a library for working with internal EEPROM which makes writing and reading data very easy. Library function `EEPROM_Read` reads data from a specified address, while library procedure `EEPROM_Write` writes data to the specified address.

Note: Be aware that all interrupts will be disabled during execution of `EEPROM_Write` routine (GIE bit of `INTCON` register will be cleared). Routine will set this bit on exit. Ensure minimum 20ms delay between successive use of routines `EEPROM_Write` and `EEPROM_Read`. Although EEPROM will write the correct value, `EEPROM_Read` might return undefined result.

In our following example, we will write a sequence of numbers to successive locations in EEPROM. Afterwards, we'll read these and output to `PORTB` to verify the process.

```

program EEPROM_test

dim i as byte
dim j as byte

main:

    TRISB = 0
    for i = 0 to 20
        EEPROM_Write(i, i + 6)
    next i

    Delay_ms(30)

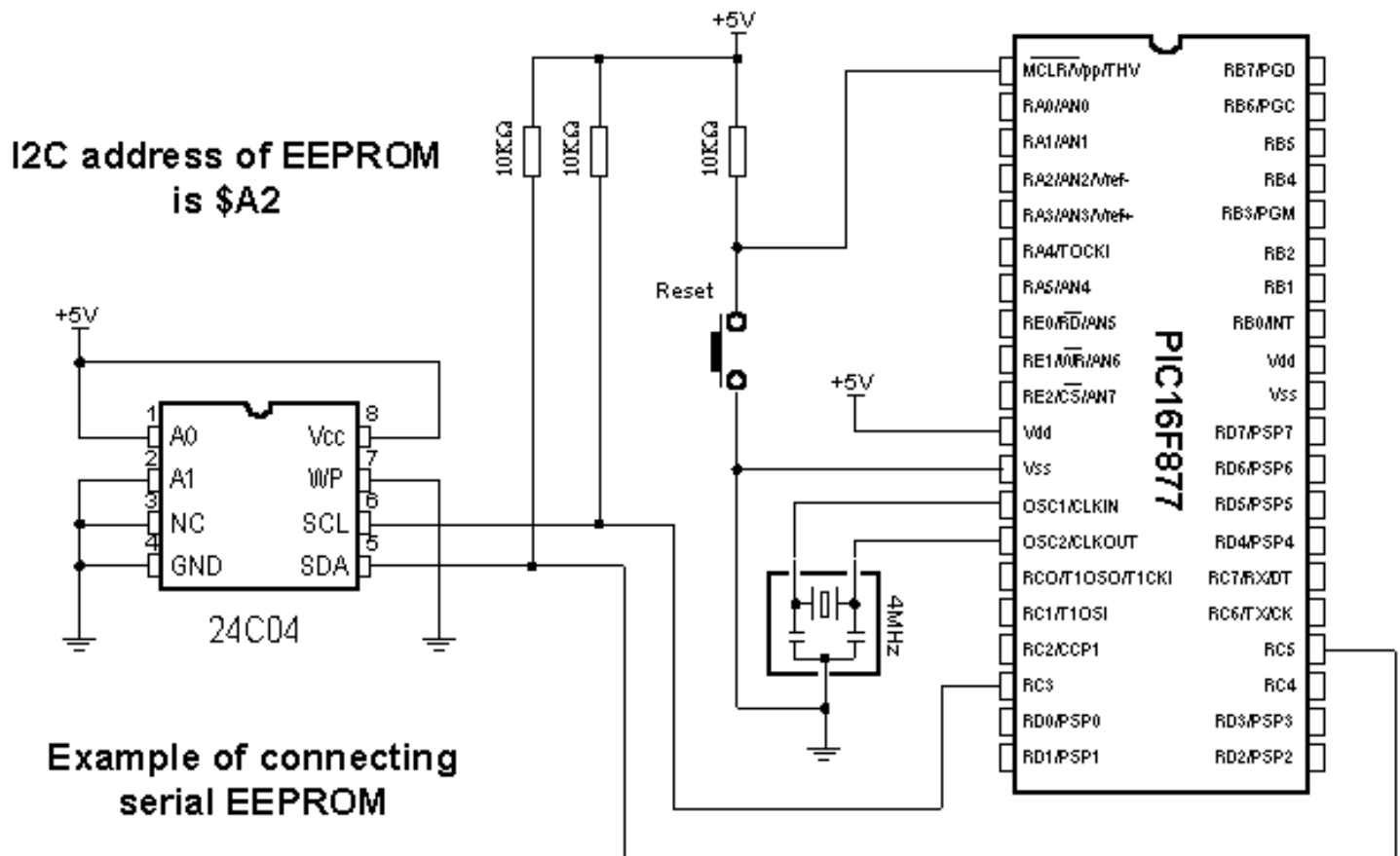
    for i = 0 to 20
        PORTB = EEPROM_Read(i)
        for j = 0 to 200
            Delay_us(500)
        next j
    next i

end.

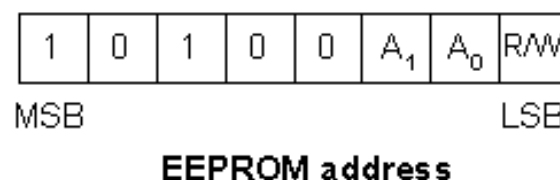
```

8.1.2 Serial EEPROM

Occasionally, our needs will exceed the capacity of PIC's internal EEPROM. When we need to store a larger amount of data obtained by PIC, we have an option of using external serial EEPROM. Serial means that EEPROM uses one of the serial protocols (I2C, SPI, microwire) for communication with microcontroller. In our example, we will work with EEPROM from 24Cxx family which uses two lines and I2C protocol for communication with MCU.



Serial EEPROM connects to microcontroller via SCL and SDA lines. SCL line is a clock for synchronizing the transfer via SDA line, with frequency going up to 1MHz.



I2C communication allows connecting multiple devices on a single line. Therefore, bits A_1 and A_0 have an option of assigning addresses to certain I2C devices by connecting the pins A_1 and A_0 to the ground and +5V (one I2C line could be EEPROM on address \$A2 and, say, real time clock PCF8583 on address \$A0). R/W bit of address byte selects the operation of reading or writing data to memory. More detailed data on I2C communication can be found in the technical documentation of any I2C device.

Our following program sends data to EEPROM at address 2. To verify transfer, we'll read data via I2C from EEPROM and send its value to PORTD. For more information on I2C Library consult Chapter 5: Built-in and Library Routines.

```

program EEPROM_test

dim EE_adr as byte
dim EE_data as byte
dim jj as word

main:
    I2C_init(100000)      ' Initialize full master mode
    TRISD = 0            ' PORTD is output
    PORTD = $ff          ' Initialize PORTD
    I2C_Start            ' Issue I2C start signal
    I2C_Wr($a2)           ' Send byte via I2C(command to 24c02)
    EE_adr = 2
    I2C_Wr(EE_adr)        ' Send byte(address of EEPROM)
    EE_data = $aa
    I2C_Wr(EE_data)       ' Send data(data that will be written)
    I2C_Stop             ' Issue I2C stop signal

    for jj = 0 to 65500  ' Pause while EEPROM writes data
        nop
    next jj

    I2C_Start            ' Issue I2C start signal
    I2C_Wr($a2)           ' Send byte via I2C
    EE_adr = 2
    I2C_Wr(EE_adr)        ' Send byte(address for EEPROM)
    I2C_Repeated_Start   ' Issue I2C repeated start signal
    I2C_Wr($a3)           ' Send byte(request data from EEPROM)
    EE_data = I2C_Rd(1)   ' Read the data
    I2C_Stop             ' Issue I2C_Stop signal
    PORTD = EE_data       ' Print data on PORTD
noend:                  ' Endless loop
    goto noend

```

```
end.
```

8.2 Flash Memory

Flash memory is a form of EEPROM that allows multiple memory locations to be erased or written in one programming operation. Normal EEPROM only allows one location at a time to be erased or written, meaning that Flash can operate at higher effective speeds when the systems using it read and write to different locations at the same time.

Flash memory stores information on a silicon chip in a way that does not need power to maintain the information in the chip. This means that if you turn off the power to the chip, the information is retained without consuming any power. In addition, Flash offers fast read access times and solid-state shock resistance. These characteristics make it very popular for microcontroller applications and for applications such as storage on battery-powered devices like cell phones.

Many modern PIC microcontrollers utilize Flash memory, usually in addition to normal EEPROM storage chip. Therefore, BASIC provides a library for direct accessing and working with MCU's Flash. **Note:** Routines differ for PIC16 and PIC18 families, please refer to Chapter 5: Built-in and Library Routines.

The following code demonstrates use of Flash Memory library routines:

```
' for PIC18

program flash_pic18_test

const FLASH_ERROR = $FF
const FLASH_OK    = $AA

dim toRead as byte
dim i as byte
dim toWrite as byte[64]

main:
    TRISB = 0                ' PORTB is output
    for i = 0 to 63          ' initialize array
        toWrite[i] = i
```

```

next i
    Flash_Write($0D00, toWrite)      ' write contents of the
array to the address 0x0D00
    ' verify write
    PORTB = 0                        ' turn off PORTB
    toRead = FLASH_ERROR             ' initialize error state

for i = 0 to 63
    toRead = Flash_Read($0D00+i)      ' read 64 consecutive
locations starting from 0x0D00
    if toRead <> toWrite[i] then    ' stop on first error

        PORTB = FLASH_ERROR          ' indicate error
        Delay_ms(500)

    else
        PORTB = FLASH_OK             ' indicate there is no error
    end if
next i
end.

```

For PIC16 family, the corresponding code looks like this:

```

' for PIC16

program flash_pic16_test

const FLASH_ERROR = $FF
const FLASH_OK    = $AA

dim toRead as word
dim i as word

main:
    TRISB = 0                        ' PORTB is output
    for i = 0 to 63

```



```

Flash_Write(i+$0A00, i)           ' write the value of i
starting from the address 0x0A00
next i
' verify write
PORTB = 0                         ' turn off PORTB
toRead = FLASH_ERROR              ' initialize error state
for i = 0 to 63
    toRead = Flash_Read($0A00+i)  ' Read 64 consecutive
locations starting from 0x0A00
    if toRead <> i then            ' Stop on first error
        i = i + $0A00            ' i contains the address of
the erroneous location
        PORTB = FLASH_ERROR      ' indicate error
        Delay_ms(500)
    else
        PORTB = FLASH_OK         ' indicate there is no error
    end if
next i
end.

```

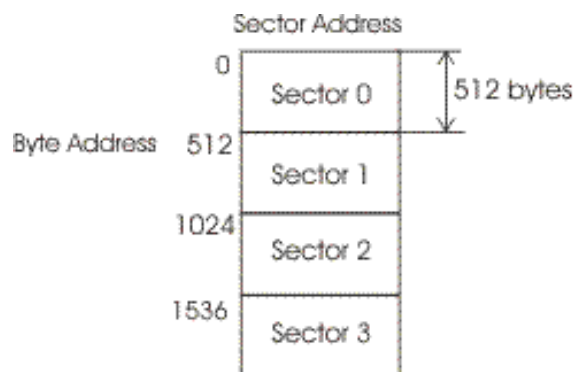
8.3 Compact Flash

Compact Flash (CF) was originally a type of data storage device, used in portable electronic devices. As a storage device, it typically uses Flash memory in a standardized enclosure. At present, the physical format is used in handheld and laptop computers, digital cameras, and a wide variety of other devices, including desktop computers. Great capacity (8MB ~ 8GB, and more) and excellent access time of typically few microseconds make them very attractive for microcontroller applications.

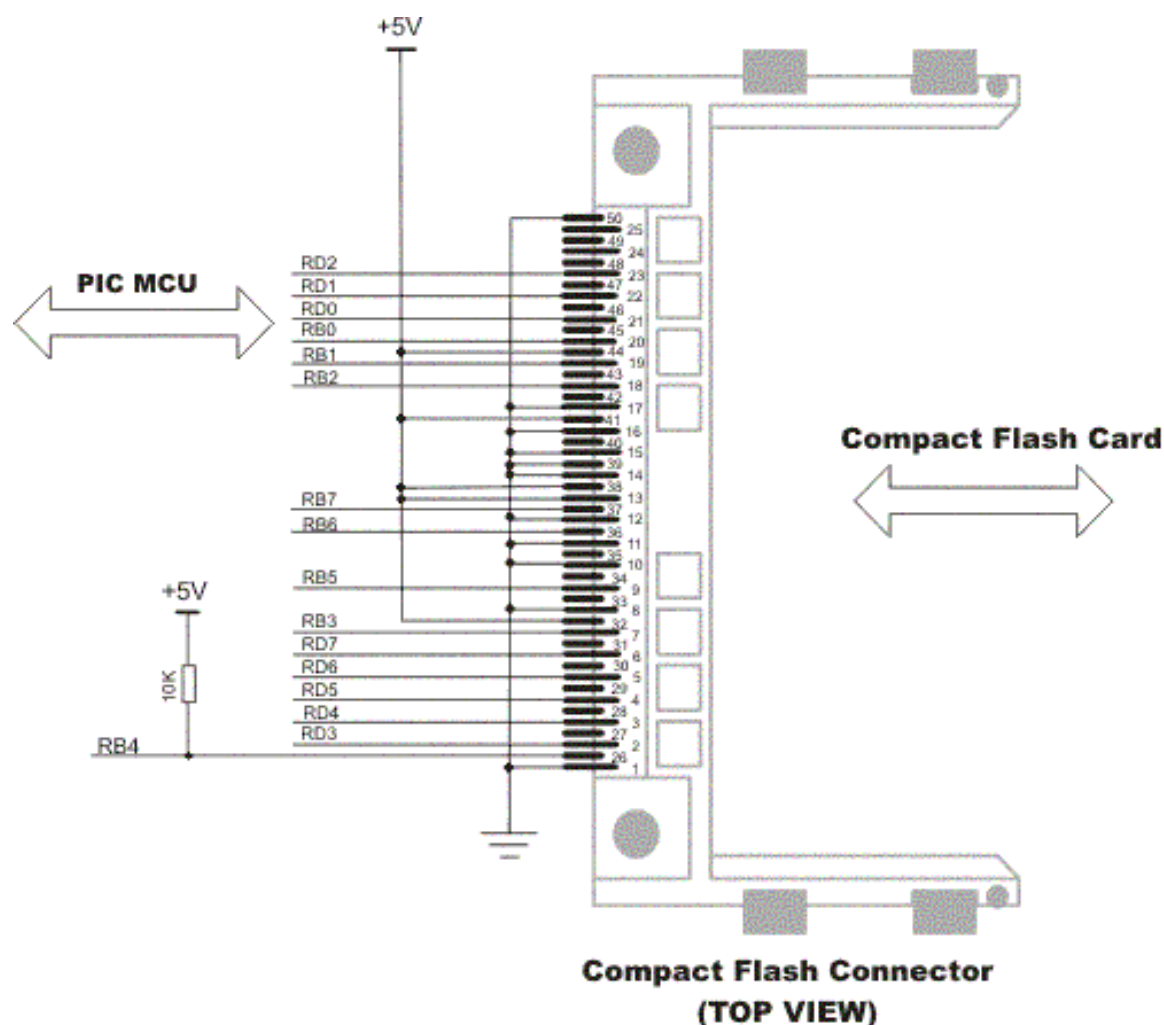
Flash memory devices are non-volatile and solid state, and thus are more robust than disk drives, consuming only about 5% of the power required by small disk drives. They operate at 3.3 volts or 5 volts, and can be swapped from system to system. CF cards are able to cope with extremely rapid changes in temperature – industrial versions of flash memory cards can operate at a range of -45°C to +85°C.

BASIC includes a library for accessing and handling data on Compact Flash card. In CF card, data is divided into sectors, one sector usually comprising 512 bytes (few older models have sectors of 256B). Read and write operations are not performed directly, but successively through 512B buffer. These

routines are intended for use with CF that have FAT16 and FAT32 file system. **Note:** routines for file handling (CF_File_Write_Init, CF_File_Write_Byte, CF_File_Write_Complete) can only be used with FAT16 file system, and only with PIC18 family!



File accessing routines can write file. File names must be exactly 8 characters long and written in uppercase. User must ensure different names for each file, as CF routines will not check for possible match. Before write operation, make sure you don't overwrite boot or FAT sector as it could make your card on PC or digital cam unreadable. Drive mapping tools, such as Winhex, can be of a great assistance.



Here's an example for using Compact Flash card from BASIC. A set of files is written on CF card. This can be checked later by plugging the CF card on a PC or a digital camera. Observe the way the file is being written:

- First, write-to-file is initialized, telling to PIC that all consecutive CF_File_Write_Byte instructions will write to a new file;
- Then, actual write of data is performed (with CF_File_Write_Byte);
- Finally, finish of write-to-file cycle is signallized with call to CF_File_Write_Complete routine. At that moment, the newly created file is given its name.

```

program CompactFlash_File
' for PIC18

dim i1 as word
dim index as byte
dim fname as char[9]
dim ext as char[4]

sub procedure Init
    TRISC = 0                                ' PORTC is output. We'll
use it only to signal
                                           ' end of our program.
    CF_Init_Port(PORTB, PORTD)              ' Initialize ports
do
    nop
loop until CF_DETECT(PORTB) = true          ' Wait until CF card is
inserted
    Delay_ms(50)                             ' Wait for a while until
the card is stabilized
end sub                                     ' i.e. its power supply
is stable and CF card
                                           ' controller is on

main:
    ext = "txt"                               ' File extensions will be
"txt"
    index = 0                                ' Index of file to be
written

```

```

while index < 5
    PORTC = 0
    Init
    PORTC = index
    CF_File_Write_Init(PORTB, PORTD)
Initialization for writing to new file
    i1 = 0
    while i1 < 50000
        CF_File_Write_Byte(PORTB,PORTD,48+index)  ' Writes 50000
        bytes to file
        inc(i1)
    wend
    fname = "RILEPROX"  ' Must be 8
    character long in upper case
    fname[8] = 48 + index  ' Ensure that
    files have different name
    CF_File_Write_Complete(PORTB,PORTD, fname, ext)  ' Close
    the file
    Inc(index)
wend
    PORTC = $FF

end.

```

If you do not wish to use your CF card in PCs and digicams but rather as a simple storage device for your PIC MCU only, you can then ignore the entire FAT system and store data directly to CF memory sectors:

```

program cf_test

dim i as word

main:
    TRISC = 0  ' PORTC is output
    CF_Init_Port(PORTB,PORTD)  ' Initialize ports

```

```

do
    nop
    loop until CF_Detect(PORTB) = true      ' Wait until CF
card is inserted

    Delay_ms(500)
    CF_Write_Init(PORTB, PORTD, 590, 1)      ' Initialize write
at sector address 590
                                           '   of   1 sector
(512 bytes)
    for i = 0 to 511                        ' Write 512 bytes
to sector (590)
        CF_Write_Byte(PORTB, PORTD, i + 11)
    next i
    PORTC = $FF
    Delay_ms(1000)
    CF_Read_Init(PORTB, PORTD, 590, 1)      ' Initialize write
at sector address 590
                                           '   of   1 sector
(512 bytes)
    for i = 0 to 511                        ' Read 512 bytes
from sector (590)
        PORTC = CF_Read_Byte(PORTB, PORTD)  '   and display
it on PORTC
        Delay_ms(1000)
    next i
end.

```

PIC, PIC, PICmicro, and MPLAB are registered and protected trademarks of the Microchip Technology Inc. USA. Microchip logo and name are the registered tokens of the Microchip Technology. mikroBasic is a registered trade mark of mikroElektronika. All other tokens mentioned in the book are the property of the companies to which they belong.

mikroElektronika © 1998 - 2004. All rights reserved. If you have any questions, please contact our **office**.